Trabalho Prático: TCP e UDP

Aluno em Graduação da Universidade Federal de Ouro Preto do curso Ciência da Computação:

Halliday Gauss Costa dos Santos

. **Matrícula:** 18.1.4093.

Área: Redes de Computadores.

Introdução:

A camada de Transporte é a camada responsável por realizar o transporte dos pacotes de dados por meio da abstração das camadas inferiores (Física, Enlace e Rede) e ainda fornecer uma abstração para a camada de aplicação. O transporte é feito por meio de protocolos, TCP e UDP, e o primeiro é um serviço orientado a conexão que garante a entrega de todos os dados enviados, já o segundo é um serviço não orientado a conexão, e, portanto, não há confiabilidade de que os dados enviados chegarão ao destino. Este trabalho prático consiste na implementação dos protocolos supracitados na linguagem de programação Python e na análise dos resultados encontrados.

Desenvolvimento:

TCP:

A primeira parte do trabalho consiste na implementação do protocolo TCP, o TCP é um serviço orientado a conexão que garante a entrega dos dados enviados, para isso é necessário estabelecer uma conexão entre o receptor e o emissor e estabelecer uma rota utilizando circuitos virtuais afim de realizar o transporte dos dados de maneira que garanta o melhor caminho na rede, a ordem dos dados, e a confiabilidade dos mesmos.

Para desenvolver o trabalho foi necessário a criação de um Servidor que permite a conexão de vários Clientes que enviam dados ao mesmo, afim de obter a resposta de uma tarefa sobre os dados enviados. A tarefa executada pelo Servidor é a de contagem de vogais, consoantes e números dos dados recebidos, o Servidor deve retornar a quantidade de cada caractere alfanumérico e caso não encontre nenhum caractere desse tipo, é retornado uma mensagem de erro.

Tanto o Servidor TCP como o Cliente TCP foram implementados como códigos distintos em Python, os códigos são respectivamente ‘servidorTCP.py’ e ‘clienteTCP.py’.

Logo abaixo será mostrado o passo a passo do funcionamento dos códigos supracitados:

‘clienteTCP.py’

```python
import socket
import time
import sys

def lerArquivo():
    """Funcao que leh os dados de arquivo de entrada e retorna cada linha lida"""

    #se o nome do arquivo nao for passado ao executar o programa, tentarah abrir o arquivo "modelo_entrada.txt"
    if len(sys.argv) > 1:
        nomeDoArquivo = sys.argv[1]
    else:
        nomeDoArquivo = "modelo_entrada.txt"

    #ler dados relevantes do arquivo
    dadosDoArquivo = []
    with open(nomeDoArquivo) as arquivo:
        #para cada linha do arquivo
        for linha in arquivo:
            #remover linhas vazias
            if linha != "\n":
                #remover comentarios e adicionar como dados validos
                if len(linha) >1:
                    if linha[0] != "/" and linha[1] != "/":
                        #remove o caractere '\n' que toda linha possui no final
                        dadosDoArquivo.append(linha.replace("\n", ""))
                else:
                    dadosDoArquivo.append(linha.replace("\n", ""))

    return dadosDoArquivo
```

Primeiramente algumas bibliotecas são importadas, a biblioteca socket (para trabalhar com sockets), a biblioteca time para trabalhar com tempo, e a biblioteca sys para trabalhar com o sistema operacional.

A função ‘lerArquivo’ é responsável por ler os dados úteis de um arquivo de entrada, ou seja, ignora os comentários, e por fim retorna esses dados. Essa função é necessária pois esses dados relevantes que estão contidos no arquivo que serão enviados ao Servidor.

Inicialmente, ao executar o código do Cliente, se o nome do arquivo for passado como parâmetro, então esse será o nome arquivo que contém os dados que serão enviados ao Servidor, caso contrário o arquivo de entrada será

‘modelo_entrada.txt’. Nesse arquivo deve possuir inicialmente a quantidade de dados que serão enviados ao servidor e em seguida os dados propriamente ditos.

Os dados úteis são lidos e armazenados numa lista que será retornada pela função. A primeira posição dessa lista possui a quantidade de dados e as posições restantes contém as string’s’ que serão enviadas ao Servidor.

```
#ler dados do arquivo
dadosDoArquivo = lerArquivo()

#atribuir quantidade de dados
quantidadeDeDados = int(dadosDoArquivo[0])

#atribuir dados
dados = dadosDoArquivo[1::]

#definir o host ("IP") e a porta
host = 'localhost'
porta = 8080

#tamanho do buffer para receber dados
buffer = 1024

#criar um socket
cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#conectar a um servidor
try:
    cliente.connect((host, porta))
except ConnectionRefusedError:
    print("\nO servidor nao estah aceitando conexoes !\nPor favor tente novamente em alguns segundos.")
    exit()

cliente.sendall("Eu sou um cliente, estou me conectando ao servidor".encode())

#receber ACK

resposta = cliente.recv(buffer).decode()
print(resposta)
```

Após a definição da função ‘lerArquivo’ começa realmente a execução do código, a função ‘lerArquivo’ é chamada e os dados úteis do arquivo de entrada são lidos. Na sequência é definido o host e a porta que o Cliente deve se conectar, e o tamanho do buffer para receber dados. Então, um socket é criado utilizando AF_INET (Protocolo IPv4, protocolo padrão da camada de rede) e SOCK_STREAM que é para trabalhar com protocolo TCP.

O Cliente tenta se conectar ao servidor, caso não consiga é impresso na tela uma mensagem de erro, mas se possível, o Cliente envia uma mensagem ao

Servidor e espera um ACK confirmando o recebimento da mensagem do Cliente, essa resposta é impressa na tela.

```
#envia a quantidade de dados que serao enviados
cliente.sendall(str(quantidadeDeDados).encode())
time.sleep(0.2)

#para cada dado
for dado in dados:
    #envia o tamanho dos dados que serao enviados
    cliente.sendall(str(len(dado)).encode())
    time.sleep(0.2)

    #depois envia os dados
    cliente.sendall(dado.encode())
    time.sleep(0.2)


print("\nEsperando Resposta do Servidor !")
```

O Cliente envia o número de solicitações que ele irá fazer e espera um tempo de 0,2 segundos(timeout) para começar a enviar os dados. Em seguida, para cada dado a ser enviado é também despachado o tamanho do dado e depois de um tempo de espera de 0,2 segundos o dado é codificado em formato binário e é enviado ao Servidor. Após todo esse procedimento é impresso na tela que o Cliente está esperando as respostas do Servidor.

```
#recebe os dados do servidor imprime na tela
respostas = []
for i in range(quantidadeDeDados):
    resposta = cliente.recv(buffer).decode()
    print(resposta)
    respostas.append(resposta)

#mensagem de terminar a execucao
resposta = cliente.recv(buffer).decode()
print(resposta)

#escrever os resultados no arquivo "modelo_saida.txt"
with open("modelo_saida.txt", 'w') as arquivo:
    for resposta in respostas:
        arquivo.write(resposta + "\n")
```

Finalmente, o Cliente fica no aguardo das respostas do Servidor, após receber uma resposta imprime a mesma na tela e armazena numa lista de respostas.

Por último, espera-se que o Servidor mande uma mensagem para terminar a execução, e as respostas das contagens são salvas no arquivo ‘modelo_saida.txt’.

‘servidorTCP.py’

```python
import socket
import threading
import math
from collections import Counter
import time


def eh_numero(caractere):
    """funcao que recebe um caractere e retorna true se ele eh um numero e false caso contrario"""
    if 48 <= ord(caractere) <= 57:
        return True

    return False


def eh_vogal(caractere):
    """funcao que recebe um caractere e retorna true se ele eh uma vogal e false caso contrario"""
    if caractere in "aeiouAEIOU":
        return True

    return False

def eh_consoante(caractere):
    """funcao que recebe um caractere e retorna true se ele eh uma consoante e false caso contrario"""
    if caractere in "bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ":
        return True

    return False
```

No código ‘servidorTCP.py’ é importado os módulos: ‘socket’ para trabalhar com sockets, ‘threading’ para trabalhar com threads, ‘Counter’ para trabalhar com contagem de caracteres, e time para trabalhar com tempo.

É definido também as funções: ‘eh_numero’ que recebe um caractere e retorna verdadeiro se ele é um número e falso caso contrário, ‘eh_vogal’ que recebe um caractere e retorna verdadeiro se ele é uma vogal e falso caso contrário, ‘eh_consoante’ que recebe um caractere e retorna verdadeiro se ele é uma consoante e falso caso contrário.

```python
def contarCaracteres(dado):
    """conta os caracteres alfa numericos e retorna uma string com a contagem de consoantes, vogais e numeros"""
    contagem = dict(Counter(dado))

    consoante = 0
    vogais = 0
    numeros = 0

    #confere se os caracteres sao vogais, consoantes, ou numeros e aumenta a contagem de cada tipo de caractere
    for caractere in contagem.keys():
        if eh_consoante(caractere):
            consoante += contagem[caractere]
        elif eh_vogal(caractere):
            vogais += contagem[caractere]
        elif eh_numero(caractere):
            numeros += contagem[caractere]

    #se nao teve nenhuma vogal, consoante ou numeros retorna erro
    if consoante == vogais == numeros == 0:
        return "erro"

    return f'C={consoante};V={vogais};N={numeros}'
```

A função ‘contarCaracteres’ recebe um dado e utiliza o módulo Counter nesse dado, o que retornará um dicionário em que as chaves são os caracteres e os valores dessas chaves são as vezes que as chaves aparecem no dado. E para cada caractere presente na chave do dicionário é adicionado ao contador de quantidade adequado o valor da chave. Por exemplo, se a chave ‘a’ é uma vogal, então é incrementado ao contador de vogal o valor do dicionário com a chave ‘a’.

Caso não seja encontrado nenhum caractere alfanumérico, ou seja, o número de consoantes, vogais e números é igual a 0 no dado, então é retornado a string ‘erro’. Caso contrário é retornado uma string que representa a quantidade de consoantes, vogais e números encontrados.

```python
def lidarComCliente(socketCliente):
    """funcao que relizarah a execucao de contagem de caracteres de cada solicitacao de cada cliente"""

    endereco = addr

    #recebe mensagem de entrada do cliente
    solicitacao = socketCliente.recv(buffer).decode()
    print("Recebido: ", solicitacao)
    print("\n-------------------------\n")

    #envia mensagem ACK para cliente
    socketCliente.sendall(f'\nMensagem destinada ao cliente {endereco[0]} {endereco[1]} \nACK ! Recebido pelo servidor !\n'.encode())
```

A função ‘lidarComCliente’ é responsável por executar todo o serviço requerido pelo Cliente dado que o mesmo já esteja conectado ao Servidor. Inicialmente é recebido o endereço do Cliente já conectado e é aguardado o recebimento de uma mensagem do Cliente para saber se a conexão está

funcionando. Após receber a mensagem ela é impressa na tela e um ACK é enviado ao Cliente avisando que a mensagem foi recebida com sucesso.

```
#executar servico
#ler a quantidade de dados que serao enviados
quantidadeDeDados = int(socketCliente.recv(buffer).decode())

#receber cada um dos dados
dados = []
for i in range(quantidadeDeDados):
    dado = ""
    #receber tamanho do dado
    tamanhoDoDado = int(socketCliente.recv(buffer).decode())

    #repete do loop 'n' vezes, quem 'n' eh o numero de vezes que o dado completo cabe no buffer
    for j in range(math.ceil(tamanhoDoDado/buffer)):
        dado = dado + socketCliente.recv(buffer).decode()

    dados.append(dado)
```

Ainda na mesma função, é recebido do Cliente a quantidade de solicitações que ele enviará, e para cada solicitação é recebido o tamanho da solicitação. Pelo tamanho de cada solicitação é calculado quantas vezes o Servidor vai receber uma mesma solicitação, para isso deve-se saber quantas vezes o buffer cabe nos dados. Obs: sempre deve arredondar para o inteiro superior mais próximo, por exemplo, se o buffer tem tamanho 50 e o dado tem tamanho 51 então é necessário chamar a função de receber dados do Cliente duas vezes. Em seguida, o dado decodificado será adicionado em uma lista de dados.

```
#contar caracteres de todos os dados enviados pelo cliente
contagem = list(map(contarCaracteres, dados))

#enviar resultados para cliente
for cont in contagem:
    socketCliente.sendall(cont.encode())
    time.sleep(0.5)

#solicitar o termino da execucao
socketCliente.sendall("Termine a execucao !".encode())

print("Cliente ", endereco, "atendido !")
#fechar conexao
socketCliente.close()
```

Continuando a descrição da implementação da função ‘lidarComCliente’, é calculado a contagem de caracteres alfanuméricos de cada dado, e cada contagem é adicionada na lista chamada de ‘contagem’, e cada item dessa lista é enviado para o Cliente com um tempo de espera de 0.5 segundos antes de cada envio para que os dados cheguem separados por solicitação.

Seguidamente o Servidor vai enviar uma mensagem para o Cliente para informar o término da conexão. É impresso na tela que o Cliente foi atendido e finalmente a conexão é fechada.

```
ip = 'localhost'
porta = 8080

buffer = 50

#AF_INET -> servico IPv4, SOCK_STREAM -> Protocolo Tcp
servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#cria o servidor com o endereco de ip e porta esepecificados acima
servidor.bind((ip, porta))

MAXIMOCONEXOES =2

#determinar o numero maximo de conexoes simultaneas
servidor.listen(MAXIMOCONEXOES)

print("Escutando", ip, porta)
```

Nesta parte começa a execução do código, primeiramente são definidos um host (IP), uma porta e o tamanho do buffer, e então um Servidor é inicializado tendo como características os protocolos IPv4 (AF_INET) e TCP(SOCK_STREAM). Na sequência o Servidor é criado na porta e host (endereço IP) especificado (bind), e no código acima o Servidor está aceitando no máximo duas conexões (listen), e é printado na tela que o Servidor está no modo escuta, ou seja, esperando conexões.

```
NaoDeuTimeOut = True

#enquanto nao der um timeout ou seja, enquanto chegar conexoes repete o laco abaixo
while NaoDeuTimeOut:
    try:
        #timeout de 60 segundos para que chegue uma conexao
        servidor.settimeout(60)
        cliente, addr = servidor.accept() #retorna o cliente e as informacoes da conexao
        servidor.settimeout(None)

    except ConnectionResetError:
        print("Deu o time out, nenhuma nova conexao")
        NaoDeuTimeOut = False
        # espera 30 segundos antes de terminar a execucao do servidor
        # para dar tempo de alguns cliente terminarem de executar
        # neste momento o sevidor nao aceita mais conexoes
        time.sleep(30)
        break
    except socket.timeout:
        print("Deu o time out, nenhuma nova conexao")
        NaoDeuTimeOut = False
        # espera 30 segundos antes de terminar a execucao do servidor
        # para dar tempo de alguns cliente terminarem de executar
        # neste momento o sevidor nao aceita mais conexoes
        time.sleep(30)
        break

    print("Conexao aceita de:", addr[0], addr[1])

    #cria uma thread para cada cliente aceito
    socketCliente = threading.Thread(target=lidarComCliente, args=(cliente,))

    socketCliente.start()
```

Imediatamente entra-se em um loop enquanto não houver timeout, nesse bloco o Servidor fica esperando conexões por 60 segundos, caso não apareça nenhum Cliente querendo se conectar nesse intervalo o Servidor sai do modo de escuta e espera 30 segundos para que os Clientes já conectados terminem seu processamento, e depois encerra o seu próprio processamento.

Caso um Cliente se conecte ao Servidor então não haverá mais o timeout, e é impresso na tela que uma conexão foi aceita e os detalhes da conexão, em seguida é criada uma thread para que o Cliente possa ser atendido (função 'lidarComCliente') enquanto o Servidor espera por novas conexões.

Para executar os códigos basta digitar no prompt de comando:

- **python clienteTCP.py** para executar o Cliente
- **python servidorTCP.py** para executar o Servidor

Cabe ressaltar que o arquivo de entrada deve ser definido antes de executar o Cliente.

Um arquivo de teste e os códigos (Cliente TCP e Servidor TCP) estão dentro da pasta 'tcp' assim como a saída resultante, e dentro da pasta 'tcp' tem outra pasta chamada test, que possui o código do Cliente e outro arquivo de entrada teste diferente, assim como a saída resultante. Dessa forma basta executar o Servidor e o Cliente na pasta tcp e o outro Cliente na pasta test para comprovar a veracidade da solução.

UDP:

A segunda parte do trabalho consiste na implementação do protocolo UDP, o UDP é um serviço que não é orientado a conexão, portanto, não garante a entrega dos dados enviados e nem a ordem, utiliza uma rede de datagramas e o Cliente simplesmente envia o pacote pela rede sem se preocupar em manter uma conexão com o Servidor. Como não há garantia que os dados irão chegar ao destino, então, após um tempo sem resposta, tanto o Servidor quanto o Cliente devem encerrar os seus procedimentos.

Como uma rota não é estabelecida no protocolo UDP o pacote enviado pela rede pode tomar um caminho ruim e chegar com atraso ou se perder, para simular isso, foi colocado um tempo de espera aleatório antes enviar um pacote.

Tanto o Servidor UDP como o Cliente UDP foram implementados como códigos distintos em Python, os códigos são respectivamente 'servidorUDP.py' e 'clienteUDP.py'.

Logo abaixo será mostrado o passo a passo do funcionamento dos códigos que acabaram de ser citados acima:

‘clienteUDP.py’

```python
import socket
import time
import sys
from random import randint

def lerArquivo():
    """Funcao que leh os dados de arquivo de entrada e retorna cada linha lida"""

    #se o nome do arquivo nao for passado ao executar o programa, tentarah abrir o arquivo "modelo_entrada.txt"
    if len(sys.argv) > 1:
        nomeDoArquivo = sys.argv[1]
    else:
        nomeDoArquivo = "modelo_entrada.txt"

    #ler dados relevantes do arquivo
    dadosDoArquivo = []
    with open(nomeDoArquivo) as arquivo:
        #para cada linha do arquivo
        for linha in arquivo:
            #remover linhas vazias
            if linha != "\n":
                #remover comentarios e adicionar como dados validos
                if len(linha) >1:
                    if linha[0] != "/" and linha[1] != "/":
                        dadosDoArquivo.append(linha.replace("\n", ""))
                else:
                    dadosDoArquivo.append(linha.replace("\n", ""))

    return dadosDoArquivo
```

O código ‘clienteUDP.py’ tem a mesma função ‘lerArquivo’ e os mesmos módulos que o ‘clienteTCP.py’, com exceção da função importada ‘randint’, a qual espera uma faixa de valores como parâmetro, e retorna um inteiro aleatório entre os valores escolhidos.

```
#ler dados do arquivo
dadosDoArquivo = lerArquivo()

#atribuir quantidade de dados
quantidadeDeDados = int(dadosDoArquivo[0])

#atribuir dados
dados = dadosDoArquivo[1::]

#definir o host ("IP") e a porta
host = 'localhost'
porta = 8080

#tamanho do buffer para receber dados
buffer = 1024

#criar um socket
cliente = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

print("\nEnviando dados ao Servidor...")
```

Na linha de execução principal do código, os dados úteis são lidos do arquivo de entrada e são definidos o host, a porta e o tamanho do buffer. Em seguida um socket do Cliente é criado contendo como característica os protocolos IPv4 (AF_INET) e UDP(SOCK_DGRAM). Feito isso, somente é necessário enviar os dados para o Servidor, pois quando se utiliza o protocolo UDP não é necessário estabelecer uma conexão, é necessário somente enviar o dado para o host e a porta especificada, ou seja, o endereço do Servidor.

```
respostas = []
try:
    #para cada dado a ser enviado para o servidor
    for dado in dados:
        #envia o dado
        time.sleep(randint(0,3))
        cliente.sendto(dado.encode(), (host, porta))

        #recebe o processamento do dado
        cliente.settimeout(60.0)
        resposta, endereco = cliente.recvfrom(buffer)
        # reseta o timeout
        cliente.settimeout(None)

        #imprime o resultado dado pelo servidor
        print(resposta.decode())

        #adiciona na lista de respostas
        respostas.append(resposta.decode())
except ConnectionResetError:
    print("Deu o time out e nao chegou mais dados")
except socket.timeout:
    print("Deu o time out e nao chegou mais dados")

#fecha o socket
cliente.close()
```

Continuando o procedimento do Cliente, para cada dado a ser enviado é esperado um tempo aleatório que pode variar de 0 até 3 segundos antes de envia-lo ao Servidor, isso é feito para simular os atrasos que podem ocorrer durante a transmissão. Em seguida, é esperado a resposta do processamento do dado pelo Servidor durante 60 segundos, caso a resposta não chegue, sairá do loop e imprimirá a mensagem informando que nem todos os dados chegaram. Caso receba a resposta do Servidor, então a resposta é decodificada, impressa na tela e adicionada numa lista de resposta. Após sair do loop o Cliente fecha o socket.

```
#escrever os resultados no arquivo "modelo_saida.txt"
with open("modelo_saida.txt", 'w') as arquivo:
    for resposta in respostas:
        arquivo.write(resposta + "\n")
```

Por fim, as respostas recebidas, completas ou não, são escritas no arquivo ‘modelo_saida.txt’.

‘servidorUDP.py’

```
import socket
from collections import Counter
import time
from random import randint
import threading


def eh_numero(caractere):
    """funcao que recebe um caractere e retorna true se ele eh um numero e false caso contrario"""
    if 48 <= ord(caractere) <= 57:
        return True

    return False


def eh_vogal(caractere):
    """funcao que recebe um caractere e retorna true se ele eh uma vogal e false caso contrario"""
    if caractere in "aeiouAEIOU":
        return True

    return False

def eh_consoante(caractere):
    """funcao que recebe um caractere e retorna true se ele eh uma consoante e false caso contrario"""
    if caractere in "bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ":
        return True

    return False
```

```python
def contarCaracteres(dado):
    """conta os caracteres alfa numericos e retorna uma string com a contagem de consoantes, vogais e numeros"""
    contagem = dict(Counter(dado))

    consoante = 0
    vogais = 0
    numeros = 0

    #confere se os caracteres sao vogais, consoantes, ou numeros e aumenta a contagem de cada tipo de caractere
    for caractere in contagem.keys():
        if eh_consoante(caractere):
            consoante += contagem[caractere]
        elif eh_vogal(caractere):
            vogais += contagem[caractere]
        elif eh_numero(caractere):
            numeros += contagem[caractere]

    #se nao teve nenhuma vogal, consoante ou numeros retorna erro
    if consoante == vogais == numeros == 0:
        return "erro"

    return f'C={consoante};V={vogais};N={numeros}'
```

O código ‘sevidorUDP.py’ utiliza os módulos mostrados nas imagens acima que já foram citados neste documento, e contém algumas funções que também estão ‘servidorTCP.py’

```python
def lidarComCliente(dado, enderecoDoCliente):
    """essa funcao vai lidar com todos uma requisicao de um cliente """

    #executar servico

    # contar caracteres do dado recebido
    contagem = contarCaracteres(dado.decode())

    # enviar resultado para cliente
    time.sleep(randint(0, 3))
    servidor.sendto(contagem.encode(), enderecoDoCliente)

    print("Cliente ", enderecoDoCliente, " atendido !")
```

Essa função vai receber um dado de um Cliente e seu endereço, e irá fazer a contagem dos caracteres alfanuméricos, na sequência vai codificar a contagem, esperar um tempo aleatório entre 0 e 3 segundos, vai enviar a resposta ao Cliente e imprimir na tela que o Cliente foi atendido.

```
buffer = 1024

ip = 'localhost'
porta = 8080

#AF_INET -> servico IPv4, SOCK_DGRAM -> Protocolo UDP
servidor = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#cria o servidor com o endereco de ip e porta esepecificados acima
servidor.bind((ip, porta))


print("Servidor Iniciado", ip, porta)
```

No código principal do ‘servidorUDP.py’ são definidos o tamanho do buffer, o IP e uma porta. Sem demora, um servidor é criado contendo como característica os protocolos IPv4 (AF_INET) e UDP(SOCK_DGRAM). E então o Servidor é inicializado na porta e IP especificado(bind).

```
#vai lidar com clientes enquanto nao houver inatividade, ou seja, timeout
while True:
    try:
        # receber dado de algum cliente qualquer
        # se nao receber nenhum dado em 60 segundos o programa eh encerrado por inatividade
        servidor.settimeout(60.0)
        dado, enderecoDoCliente = servidor.recvfrom(buffer)
        print("Dado de Cliente ", enderecoDoCliente, "recebido !")
        # resetar o timeout
        servidor.settimeout(None)

    except ConnectionResetError:
        print("Deu o time out")
        NaoDeuTimeOut = False
        break
    except socket.timeout:
        print("Deu o time out")
        NaoDeuTimeOut = False
        break

    #cria uma thread de execucao para contar os caracterers e enviar para o cliente que solicitou o servico
    #enquanto isso novas conexoes sao aceitas
    thread = threading.Thread(target=lidarComCliente, args=(dado, enderecoDoCliente,))
    thread.start()

servidor.close()
```

Dando continuidade na implementação, um loop é criado e o programa só sairá do mesmo quando passar mais de 60 segundos sem receber nenhum dado de nenhum Cliente. Mas se um dado for recebido, então uma thread é criada para

executar a função ‘lidarComCliente’, ou seja, o Servidor vai continuar esperando os pacotes chegarem durante um tempo de 60 segundos, enquanto está rodando paralelamente na thread as requisições do Cliente encima do dado enviado. Após sair do loop, ou seja, deu o timeout, o Servidor é encerrado.

Para executar os códigos relacionados ao protocolo UDP basta digitar no prompt de comando:

- **python clienteUDP.py** para executar o Cliente
- **python servidorUDP.py** para executar o Servidor

Cabe ressaltar que o arquivo de entrada deve ser definido antes de executar o Cliente.

Um arquivo de teste e os códigos (Cliente UDP e Servidor UDP) estão dentro da pasta ‘UDP’ assim como a saída resultante, e dentro da pasta ‘UDP’ tem outra pasta chamada test, que possui o código do Cliente e outro arquivo de entrada teste diferente, assim como a saída resultante. Dessa forma basta executar o Servidor e Cliente na pasta UDP e o outro Cliente na pasta test para comprovar a veracidade da solução.

Cabe ressaltar que os arquivos de testes são iguais para os ambos os protocolos, mas os arquivos de testes dentro de uma pasta de um protocolo são diferentes.

Conclusão:

Conclui-se que ambos os protocolos tem suas próprias aplicações, o TCP, por exemplo, para envio de e-mails por ser mais confiável e garantir a ordem dos pacotes e o UDP, para vídeo chamadas, já que esse serviço não garante a entrega de todos os pacotes e nem se preocupa em recuperá-los. Foi possível demostrar isso através do trabalho prático, já que na implementação, o TCP procura estabelecer uma conexão, ou seja, montar circuitos virtuais para depois enviar os pacotes, e no código com o protocolo UDP o Cliente simplesmente envia o pacote. Por conta disso, percebe-se que o UDP é mais rápido que TCP, porém se o código

for rodado em uma mesma máquina, do jeito que a implementação está, não é possível perceber tão bem isso, pois o tempo de espera aleatório para simular atraso é relativamente alto, mas sem esse tempo que foi somente para simulação o UDP seria muito mais rápido.

Este trabalho foi de suma importância para entendimento do funcionamento dos protocolos TCP e UDP e também para entender as aplicações de cada um, além disso foi possível ter a noção de como funciona realmente uma rede de computadores e a transmissão de dados entre máquinas.