Documentação do trabalho prático 3

Integrantes:
Alessandro Azevedo Duarte - Matrícula: 2017072642
Hugo Ferreira Marques Matrícula: 2018014573

É importante ressaltar que nos testes realizados utilizamos somente a topologia apresentada na documentação.
Para iniciar o programa é necessário usar o comando `python3 Server.py --addr=<ip> --update-period=<time>`

Comandos iniciais

O programa foi dividido em 2 partes:

- Thread principal - responsável pela leitura do teclado
- Thread Server - responsável por receber as mensagens UDP e enviar periodicamente a mensagem de update, essa thread é criada através de uma classe, chamada ***ServerThread***, definida no arquivo *ServerClass.py*

Os argumentos iniciais foram passados através de opções de linha de comando, ou seja, é necessário incluir os parâmetros "`--addr=<meuIP>`" e "`--update-period=<valorUpdate>`"(O parâmetro de update-period é opcional, e caso não seja definido, tem um valor de 4s). Isso é possível através do modulo getopt, demonstrado abaixo.

```
addr = None
update_period = 4 # Default value for update period

options, remainder = getopt.getopt(sys.argv[1:], 'x', ['addr=',
                                                       'update-period=',
                                                       ])


for opt, arg in options:
    if opt in ('--addr'):
        addr = arg
    elif opt in ('--update-period'):
        update_period = int(arg)

if(addr == None):
    raise ValueError('Valor ADDR nao definidos na linha de comando')
print ('ADDR   :', addr)
print ('UPDATE-PERIOD   :', update_period)
```

Além disso, a mensagem era lida através da função input() e as mensagens eram filtradas e enviadas para a função correta através de filtros Regex, definidos em um arquivo separado(Regex.py).

Estrutura Tabela Roteamento

Para maior facilidade do trabalho, cada servidor possui uma tabela de roteamento, da qual foi definida a seguinte estrutura, demonstrada como exemplo a tabela do roteador 1:

```
dictRoute1 = {
"127.0.1.1"   :   [[00,"127.0.1.1"]],
"127.0.1.2"   :   [[20,"127.0.1.5"]],
"127.0.1.3"   :   [[20,"127.0.1.5"]],
"127.0.1.4"   :   [[20,"127.0.1.5"]],
"127.0.1.5"   :   [[10,"127.0.1.5"]],
}
```

As chaves representam o IP destino, e os valores são formados por um vetor 2d, sendo composto, cada vetor de dentro por uma rota para alcançar esse IP destino, definida como seu peso e através de qual endereço tem que mandar a mensagem para se obter esse peso.

Exemplo: Para se alcançar o roteador 127.0.1.4 só existe uma rota de peso 20, e essa rota passa pelo roteador 127.0.1.5.

Mensagens JSON

Todas os tipos de mensagem estão escritas e estruturadas no arquivo JSON.py, como por exemplo para a mensagem de tipo data, recebemos o source e o destination, além do payload após a montagem também é feita a codificação da mensagem utilizando da função dumps da biblioteca json.

```
def Data(source,destination,payload):
    if (not isinstance(source, str) or not isinstance(destination, str) or not isinstance(payload, str)):
        raise TypeError("Tipo Nao suportado para a funcao")

    message = {
    "type"          : "data",
    "source"        : source,
    "destination"   : destination,
    "payload"       : payload,
    }
    message = json.dumps(message)
    return message
```

Recebimento de comandos (Regex)

Para o recebimento dos comandos também utilizamos de um regex para uma melhor interpretação dos inputs e esse regex está descrito no arquivo Regex.py, foi utilizada a biblioteca re que é a biblioteca de expressões regulares do python.

```
import re

def CheckADD(command):
    return re.search("^add [\d.]+ \d+$", command)

def CheckDEL(command):
    return re.search("^del [\d.]+$", command)

def CheckTrace(command):
    return re.search("^trace [\d.]+$", command)
```

Função *run*

A função *run* é quem inicializa a thread de envios periódicos e tem o loop para receber as mensagens, após o recebimento utilizamos da função loads para decodificação do json e de acordo com cada tipo de mensagem chamamos as respectivas funções como a ***traceRoute*** e a ***ReceiveUpdate***, se a mensagem é para o servidor e ele não precisa de repassá-la a função ***MessageForMe*** é chamada. Se a mensagem é do tipo data utilizamos a função de ***GetMenorRota*** para saber para quem devemos repassar a mensagem ou se essa rota ainda não está disponível. Após a saída do loop a conexão é encerrada.

```
def run(self):
    self.sendPeriodicThread()
    while True:
        msg, (ipRecebido, portaRecebida) = self.udp.recvfrom(BUFSZ)
        # print()
        # print("MENSAGEM RECEBIDA:",msg.decode())
        msgload = json.loads(msg.decode())
        if(msgload["type"] == "trace"):
            self.traceRoute(msg.decode())
        elif(msgload["type"] == "update"):
            self.ReceiveUpdate(msg.decode())
        else:
            if(FuncoesApoio.MessageForMe(self.myIP,msgload)): # se a mensagem for para mim,
                                                              # imprimir na tela
                print("Payload: ",msgload["payload"])
            else: # Se a mensagem nao for para mim, repassar
                if(msgload["type"] == "data"): #Repassar mensagem se for do tipo data
                    proxServ = FuncoesApoio.GetMenorRota(self.myIP,msgload["destination"],self.myRouteTable) # retorna o proximo
                                                                                                             # proximo servidor a
                                                                                                             # repassar a mensagem
                    if(proxServ == 0):
                        print("Rota nao disponivel para",msgload["destination"])
                    else:
                        self.SendMsgTo(msg.decode(),proxServ)
                        print("Enviar entao essa mensagem para:",proxServ) #


    print('Finalizando conexao do cliente')
    self.udp.close()
```

Função SendMsgTo

Essa função é a responsável por enviar a mensagem ao seu respectivo servidor de destino utilizando o protocolo UDP. É feito um cálculo para analisar para qual servidor a mensagem será enviada, e para isso é utilizada a função ***GetMenorRota***

```
def SendMsgTo(self,msg,ipDest):
    if(ipDest in self.myRouteTable):
        proxServ = FuncoesApoio.GetMenorRota(self.myIP,ipDest,self.myRouteTable) # retorna o proximo servidor a repassar a mensagem
        dest = (proxServ, 55151)
        self.udp.sendto(msg.encode(), dest)
    else:
        print("ERROR - nao eh possivel enviar esse dado pois a rota nao eh conhecida")
```

Função *Trace*:

Ao receber o comando trace, a mensagem é formatada utilizando a função TRACE do arquivo JSON e em seguida a função ***getmenorRota*** é chamada para ver qual será o próximo servidor para mandar o trace (podendo ser até mesmo o servidor final).
Quando um servidor recebe uma mensagem do tipo trace ele executa a função ***TraceRoute***, que:

- adiciona o IP do servido ao final do “hops”,
- analisa a mensagem desse tipo e:
    - caso o servidor que recebeu a mensagem seja o destino final, envia uma mensagem do tipo “data” que contem como payload a mensagem do trace
    - caso contrário, envia uma mensagem do tipo “trace” para o próximo servidor da rota (utiliza novamente a função ***getmenorRota)***

```
def TraceCommand(self,ipDest): # Formata o 1o trace e envia para o proximo servidor
    msg = JSON.Trace(self.myIP,ipDest,[self.myIP])
    proxServ = FuncoesApoio.GetMenorRota(self.myIP,ipDest,self.myRouteTable) # retorna o proximo servidor a repassar a mensagem
    if(proxServ !=0):
        self.SendMsgTo(msg,proxServ)
        # print("Enviar 1o Trace:",msg)
    else:
        print("ERROR - Server nao esta na lista, aguarde os updates, ou adicione uma rota")

def traceRoute(self,JSONmsg): # Funcao a se usar quando recebe uma mensagem do tipo trace
    msg = json.loads(JSONmsg)
    msg["hops"].append(self.myIP)
    resultJSON = json.dumps(msg)
    if(self.myIP == msg["destination"]):
        dataMsg = JSON.Data(self.myIP,msg["source"],resultJSON)
        self.SendMsgTo(dataMsg,msg["source"])# Aqui eles se invertem uma vez q o ultimo trace q recebeu e chegou ao destino precisa repassar ao de origem
    else:
        proxServ = FuncoesApoio.GetMenorRota(self.myIP,msg["destination"],self.myRouteTable) # retorna o proximo servidor a repassar a mensagem
        self.SendMsgTo(resultJSON,proxServ) # Caso nao seja o ultimo, repassar para o prox serv
```

Thread de envio periódico da mensagem de update:

A thread de envio periódico utiliza do valor timeToSend para enviar periodicamente a mensagem de update para os respectivos IP’s, como relatado acima essa variável de tempo pode ser alterada pelo comando `--update-period=<valorUpdate>`.

```
def sendPeriodicThread(self):
    # print("COMECO ENVIO PERIODICO")
    for i in self.myRouteTable:
        if(i == self.myRouteTable[i][0][1] and i != self.myIP): # Checar se eh um roteador vizinho e se nao eh ele mesmo
            # print("ENVIANDO UPDATE para:",i)
            self.sendPeriodic(i)
    # print("TERMINO ENVIO PERIODICO")
    threading.Timer(self.timeToSend, self.sendPeriodicThread).start()
```

Função *sendPeriodic*:

A função sendPeriodic recebe o ip de destino que no caso é enviado pela tabela acima enquanto ela percorre a tabela de rotas locais, como descrito nas especificações utilizamos uma verificação para que as rotas do ip local não sejam enviadas e uma função de apoio que é a ***GetMenorPesoRota*** que é bastante semelhante à função de ***GetMenorRota***, porém ela nos retorna o peso e não por qual Rota o caminho deve passar.

Utilizamos de uma verificação para a atualização dos pesos, para que quando IP precise de passar por outro até chegar no destino o valor desse peso seja adicionado na rota respectiva e se não esse valor permanece igual. Posteriormente, a mensagem de update é enviada com o dicionário auxiliar que foi criada para essas atualizações.

```
#Funcao responsavel pela mensagem de update
def sendPeriodic(self, ipDest):
    dicAux = {}
    for i in self.myRouteTable:
        aux1 = FuncoesApoio.GetMenorPesoRota(self.myIP, i, self.myRouteTable)
        aux2 = FuncoesApoio.GetMenorPesoRota(self.myIP, ipDest, self.myRouteTable)
        if (i != self.myIP and i == aux1[1]): # Não enviar a rota do meu ip e de rotas aprendidas
            if(aux1 != aux2):
                pesoDist = aux1[0] + aux2[0]
                dicAux[i] = pesoDist
            else:
                dicAux[self.myIP] = aux1[0]
    msg = JSON.Update(self.myIP, ipDest, dicAux)

    self.SendMsgTo(msg, ipDest)
```

Função *ReceiveUpdate*:

Essa função é responsável pelo recebimento da mensagem de update e adição do vetor distances na tabela local de rotas, utilizando da função AddFromTable como demonstrado acima.

```
#Função responsável pelo recebimento das mensagens de update
def ReceiveUpdate(self,JSONmsg):
    msg = json.loads(JSONmsg)
    dicAux = msg["distances"]
    for i in dicAux:
        self.AddFromTable(i,msg["source"], dicAux[i])
    print(self.myRouteTable)
```

Função AddFromTable:

Essa função é uma das mais importantes do programa, ela é responsável por adicionar a rota - caso ela não exista - com seu peso à tabela de roteamento. Utilizada tanto para adicionar pelo comando *add* quanto para adicionar pelo recebimento dos *updates*

```
def AddFromTable(self,ipDest,ipFrom,weight): #ipFrom -> de onde a conexao vem
    if(str(ipDest) in self.myRouteTable):
        for conexao in (self.myRouteTable[str(ipDest)]):
            if(conexao[1] == ipFrom):
                self.myRouteTable[str(ipDest)].remove(conexao)
        self.myRouteTable[str(ipDest)].append([int(weight), str(ipFrom)])
    else:
        self.myRouteTable[str(ipDest)] = [[int(weight), ipFrom]]
```

Função *DelFromTable*:
Essa função deleta a rota correspondente do seu parâmetro, caso essa rota seja a única possível, ela deleta a *key* do dicionário correspondente a esse endereço IP.

```
def DelFromTable(self,ip):
    if (ip in self.myRouteTable):
        for conexao in self.myRouteTable[ip]: #Passa por todas as conexoes desse ip
            if(conexao[1] == self.myIP): # Se a conexao foi feita desse enlace, remover conexao
                self.myRouteTable[ip].remove(conexao)
        if(len(self.myRouteTable[ip]) == 0): # Se apos essa remocao, a lista estiver vazia, remover a key da lista
            self.myRouteTable.pop(ip)
```

Arquivo FuncoesdeApoio

Esse arquivo contém 3 funções de apoio para algumas lógicas do programa e utilizamos da biblioteca numpy que se ainda não estiver instalada no seu sistema pode ser facilmente adicionada utilizando o comando pip install numpy.

Função *MessageForMe*

Essa é uma função boleana que retorna true se a mensagem for para o ip local e false se ela deve ser repassada.

```
def MessageForMe(meuIP,msgReceived): # Retorna verdadeiro se a mensagem e para mim, ou falso c.c
    if(msgReceived["destination"] == meuIP): # Se for para o meu ip, armazenar, senao repassar
        return True
    else:
        return False
```

Função *GetMenorRota*

Como o próprio nome diz, por meio dessa função verificamos qual o IP da menor rota pelo qual a conexão deve ser feita entre o meu Ip e o Ip de destino, primeiro verificamos se o ip de destino está na tabela local de rotas, utilizamos da função argmin da biblioteca numpy para analisar qual o menor valor e posteriormente verificamos se esse menor valor é através deste roteador, e se for a mensagem já é enviada direto e se não ele envia qual o ip respectivo da menor rota. Se não existe uma rota disponível o valor 0 é enviado.

```
def GetMenorRota(meuIP,ipDest,dict): # Retorna o IP da menor rota, para repassar a msg
    if (ipDest in dict): # Checa se existe o ip na routeTable
        min = np.argmin(dict[ipDest],axis=0)
        if(dict[ipDest][min[0]][1] == meuIP): # Se a melhor rota for atraves deste roteador, ja enviar direto
            return ipDest
        else:                                 # C.C enviar a melhor rota
            return dict[ipDest][min[0]][1]
    return 0 # Nao existem rotas disponiveis
```

Função *GetMenorPesoRota*

Semelhante à função demonstrada acima essa função também é referente a menor rota possível, porém ela retorna o menor peso dessa rota. E se não existe uma rota do meu ip para o ip de destino ele retorna o valor 0.

```
def GetMenorPesoRota(meuIP,ipDest,dict): # Retorna o peso da menor rota, para repassar a msg
    if (ipDest in dict): # Checa se existe o ip na routeTable
        min = np.argmin(dict[ipDest],axis=0)
        return dict[ipDest][min[0]]
    return 0 # Nao existem rotas disponiveis
```