Universidade Federal de Minas Gerais

Redes de Computadores
Trabalho Prático 3
DCCRIP: Protocolo de Roteamento por Vetor de Distância

Alunos: Luciano Mota Moraes - 2016020975
Matheus Henrique Gonçalves Souza - 2016068854
Curso: Engenharia Controle e Automação

Belo Horizonte, novembro 2020.

- ## Introdução:

Foi desenvolvido em python um programa para roteador virtual DCCRIP, utilizando a porta 51511 do computador e o sistema de loopback para estabelecer as conexões dos roteadores virtuais.
O projeto consiste de uma classe chamada RoteadorVirtual que contém alguns dos métodos de execução exigidos no TP.
Cada método da classe serve para dividir a responsabilidade de função, assim sendo, as threads são executadas em certas funções para permitir que os comandos de interface sejam computados, e também permitir que as mensagens sejam escutadas e trabalhadas de acordo com a lógica requerida do TP. Abaixo temos uma estrutura simplificada de comos pensamos a aplicação e o esqueleto da classe principal utilizada em todo o programa:

```
class RoteadorVirtual(builtins.object)
    |  RoteadorVirtual(ip, periodo)
    |
    |  Methods defined here:
    |
    |  addIP(self, ip, weight, nextDest)
    |      Adiciona um IP com o peso (custo de envio) para o banco de dados do Roteador
    |
    |  calculaRota(self, ip)
    |      Calcula rota de menor peso
    |
    |  deleteIP(self, ip)
    |      Deleta um IP do banco de dados do roteador
    |
    |  enviaMsg(self, destIP, data)
    |      Envia mensagens
    |
    |  enviaUpdate(self)
    |      Envia mensagem do tipo update para os roteadores vizinhos
    |
    |  recebeUpdate(self, ipDest)
    |      Percorre dicionário IP e faz update de pesos
    |
    |  recvMsg(self)
    |      Recebe a mensagem JSON pelo socket
    |
    |  resolveMsg(self, msgJson)
    |      Trata mensagens
    |
    |  trace(self, traceMsg)
    |      Trata as mensagens de trace
```

- ## Metodologia:

Os programas foram desenvolvidos e executados em ambiente Linux (16.04 LTS e WSL no Windows) e utilizando a linguagem Python 3.8.

- **Funções Base:**

Foram definidas algumas funções essenciais para o escopo antes mesmo de começar a parte de programação em si para servir de guia ao que nós devíamos nos atentar para fazer de forma mais precisa.
Essas funções foram citadas acima no escopo e já foram brevemente explicadas e seus parâmetros de entrada definidos.
Dessas, consideramos como de fundamental importância algumas como a resolveMsg:

```
def resolveMsg(self):
        '''
        Trata mensagens
        '''
        while(1):
            msgJson = recvMsg()
            mensagem = json.load(msgJson)
            if mensagem['type'] == 'data':
                if mensagem['destination'] == self.sock.getsockname():
                   print("Mensagemde{},payload:\n{}".format(mensagem['so
urce'],mensagem['payload']))
                else:
                    self.enviaMsg(mensagem)
            if mensagem['type'] == 'trace':
                self.trace(mensagem)
```

Essa função é responsável por decodificar as mensagens recebidas via json e decidir quais as próximas atitudes a serem tomadas de acordo com o campo “type” da mensagem, de acordo com isso alguma outra função será chamada para tratar a mensagem.
Aí no código acima podemos ver que se a mensagem for do tipo ‘data’ e o próprio roteador em questão for o destinatário final, ele simplesmente irá printar a source e o payload.
Mas, caso não for, a mensagem será enviada ao próximo roteador da lista pela função “enviaMsgm” que também está sendo chamada a cima até que a mensagem chegue ao destinatário final.
E por fim, caso ela for do tipo ‘trace’ a função ‘trace’ será chamada, e dentro dela caso não seja o destinatário final o campo ‘hops’ será atualizado com mais esse roteador sendo posto na ‘rota de envio’ da mensagem, e caso for a mensagem é reconstruída e então enviada pela enviaMsg.
Abaixo vemos a implementação da “trace”:

```
104
105     def trace(self,traceMsg):
106         '''
107         Trata as mensagens de trace
108         '''
109         rotIp = self.sock.getsockname()
110         traceMsg['hops'].append(rotIp)
111         if traceMsg['destination'] == rotIp:
112             print('[log] --- trace message received')
113             mensagem['type']='data'
114             mensagem['source'] = traceMsg['destination']
115             mensagem['destination'] = traceMsg['source']
116             mensagem['payload'] = traceMsg
117             enviaMsg(mensagem)
118         else:
119             enviaMsg(traceMsg)
120         pass
```

Outra das funções que foi amplamente utilizada é a enviaMsgm, como o nome já diz ela é a responsável por enviar as mensagens entre os roteadores.

```
67
68     def enviaMsg(self,data):
69         '''
70         Envia mensagens
71         '''
72         msg = json.dump(data)
73         destIP = calculaRota(data['destination'])
74         self.sock.sendto(msg,(destIP,porta))
```

Ela recebe os dados em forma de json, e após que a rota é retornado pelo método “calculaRota” a mensagem convertida é enviada via socket para o IP de destino que foi anteriormente retornado.

E por fim a “calculaRota”:

```
121
122     def calculaRota(self,ip):
123         '''
124         Calcula rota de menor peso
125         '''
126         return self.rotas[ip][2]
```

Como tivemos problemas com implementar o “Reroteamento Imediato” e apenas o “caminho ótimo” para cada roteador era armazenado, essa função apenas responde com o caminho dado o IP pedido.

- **Funcionalidades:**

Dentro do escopo do Trabalho Prático quatro funcionalidades foram requeridas de maneira especial, estas são: *Atualizações Periódicas, Split Horizon, Roteamento imediato e Remoção de Rotas Desatualizadas.*

- Atualizações Periódicas:

Implementamos isso a partir de uma condicional que verifica um temporizador que toda vez que o tempo corrido é maior que quatro vezes menor que o período de tempo de envio de mensagens de

update uma thread é disparada e a função de update é chamada. Abaixo temos os trechos chave do código onde isso foi implementado:
Primeiro temos a variável global tempo_anterior que recebe o tempo em que o programa começa a ser executado e depois vai sendo atualizado a cada ciclo de 4 vezes período. A função time() da biblioteca time retorna o tempo atual do sistema, então subtraindo o tempo anterior temos o tempo decorrido.

```
192
193     while(1):
194         if (time.time() - tempo_anterior) > 4 *roteador.periodo:
195             tempo_anterior = time.time()
196             t2 = threading.Thread(target = roteador.enviaUpdate())
197             t2.start()
```

A função chamada pela thread é a seguinte:

```
49      def enviaUpdate(self):
50          '''
51          Envia mensagem do tipo update para os roteadores vizinhos
52          '''
53          for ip in self.rota.keys():
54              if (self.rotas[ip][2] - time.time()) > self.cosnt_time:
55                  self.rotas.pop(ip)
56
57          for ipEnv in self.rotVizinho():
58              rotasEnv = split_horizon(ip)
59              self.sock.sendto(rotasEnv,(ipEnv,porta))
60          pass
```

Ela basicamente envia uma mensagem de update a todos os roteadores vizinhos (utilizando a otimização *split horizon*, que será comentada posteriormente), fazendo com que todos sejam periodicamente atualizados.

- *Split Horizon:*

A *Split Horizon* é uma otimização que visa evitar o envio de dados redundantes de um roteador a outro. O que seriam esses dados? Bom, basicamente seria o seu próprio peso nessa topologia, por exemplo, não faz sentido o roteador A enviar para o roteador B o seu peso na rede. A implementação do método foi a seguinte:

```
141         def split_horizon(rotas,destino):
142             splitted = {}
143             splitted = self.rotas
144             for rota in self.rotas:
145                 if rota[1] == destino:
146                     splitted.pop(rota)
147             return splitted
```

Primeiramente a função gera um dicionário cópia do dicionário presente no objeto roteador que contém todas as suas rotas salvas. Feito isso um laço for itera sobre as rotas e uma comparação é feita a cada iteração, e caso o roteador de destino seja a chave do dicionário essa é retirada do dicionário auxiliar e por fim esse auxiliar é retornado pela a função e ele sim é enviado na hora de realizar os updates. Essa função é chamada para cada roteador diferente de destino.

- Reroteamento Imediato:

Essa funcionalidade foi uma das que mais nos causou dificuldades e empecilhos no desenvolvimento do TP. Ela era a respeito de não perder a comunicação com os resistores seguintes caso a topologia fosse alterada e a ligação com um dos vizinhos retirada.
Devido a bugs e outros problemas acabamos por retirá-la do projeto a fim de tornar a comunicação mais simples, mas em contrapartida tentamos incorporar mecanismos para tornar as buscas por rotas para o envio de mensagem mais rápido e preciso.
Criamos uma lista contendo os roteadores “vizinhos” e optamos apenas por gravar o peso “ótimo” pra cada comunicação com eles.

- Remoção de Rotas Desatualizadas:

Por último, a quarta funcionalidade que devíamos implementar era a remoção de rotas que não eram atualizadas a um determinado período de tempo. Para isso, inicialmente inserimos um campo de “tempo” no dicionário de rotas dos roteadores, para que assim fosse possível fazer essa comparação. Essa etiqueta de tempo também foi feita utilizando a função time(), ou seja, assim que a mensagem de update contendo os pesos era recebida, um tempo era atribuído a ela.

```
46      def recebeUpdate(self,ipDest):
47          '''
48          Percorre dicionário IP e faz update de pesos
49          '''
50          for ip in self.rotaUpdate.keys():
51              if self.rotas[ip][0] > self.rotaUpdate[ip][0]:
52                  self.rotas[ip] = (self.rotaUpdate[ip][0],ipDest,time.time())
53          pass
54
55      def enviaUpdate(self):
56          '''
57          Envia mensagem do tipo update para os roteadores vizinhos
58          '''
59          for ip in self.rota.keys():
60              if (self.rotas[ip][2] - time.time()) > self.cosnt_time:
61                  self.rotas.pop(ip)
62
63          for ipEnv in self.rotVizinho():
64              rotasEnv = split_horizon(ip)
65              self.sock.sendto(rotasEnv,(ipEnv,porta))
66          pass
```

Na função enviaUpdate podemos ver que há uma comparação com self.const_time (que nada mais é do que 4 vezes o período), e que caso a temporização for maior do que isso, o dado é retirado do dicionário de rotas, pois ele é considerado obsoleto.
Já na função de recebeUpdate, podemos ver que o terceiro parâmetro a ser salvo no dicionário (na posição self.rotas[ip][2]) é a etiqueta de tempo, que contém o tempo em que ela foi atualizada, citada anteriormente.

● Conclusão:

Por fim, a maioria dos testes disponibilizados se comportaram de maneira satisfatória, principalmente o inicial que usava do arquivo “lo-addresses.sh” que era usado para popular os roteadores com IPs pré definidos. Algumas outras funções apresentaram alguns bugs em algumas circunstâncias

apesar de os métodos terem se comportado de acordo com o planejado quando testados separadamente.

Na implementação deste trabalho prático foram encontrados muitos desafios, como a manipulação e envio da codificação JSON, sendo um contato com uma nova forma de variável, que se mostrou muito interessante e com várias possíveis aplicações. Uma vez que foi esclarecido como a variável funcionava e quais suas funções disponíveis, o restante da implementação nesta parte foi facilitada.

A abstração do simulador também foi algo desafiador, uma vez em que se desenvolvia apenas um nó da topologia da rede simulada, mas que esta implementação serviria para todos os nós da rede. Logo, todos os casos já deveriam ser previstos e sido precavidos, como as recepções e envio de mensagens. Uma vez que esta abstração foi feita, o desenvolvimento também foi facilitado.

Por fim, a manipulação de threads também teve de ser bem pensada, pois apesar de já ter sido tratada nos outros dois TPs (no segundo de forma fundamental), ainda era um conceito recém aprendido, porém, na linguagem python seu uso se dá de forma mais maleável.

Também é válido salientar que o TP contribuiu bastante para a compreensão de diversos conceitos vistos durante o semestre de forma teórica