Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Redes de Computadores
Trabalho Prático 2: DCC RIP: Protocolo de Roteamento por Vetor de Distância
Alunos: Daniel Vieira da Silva Cruz
Manoel da Rocha Miranda Júnior

---

1 - Introdução

Este trabalho prático tem como objetivo a implementação de um programa que simule a execução de um roteador baseado no protocolo de roteamento por vetor de distância. O código entregue junto com este trabalho foi desenvolvido na linguagem Python, versão 3.5.2, no sistema operacional Linux. Para executar o programa, é necessário passar alguns parâmetros na linha de comando. Estes parâmetros estão descritos a seguir:

- `--addr` : Especifica o endereço do roteador que será executado. É neste endereço que o socket fará o bind e que será utilizado para enviar e receber informações da rede.
- `--update-period` : Especifica o intervalo de tempo entre as mensagens de atualização.
- `--startup` : Este parâmetro opcional especifica o nome de um arquivo texto que pode ser utilizado para executar os comandos nele presente na linha de comando do roteador. Os comandos possíveis estão descritos abaixo.

Inicialmente, foi executado o script `lo-adress.sh`, fornecido juntamente com o material do trabalho, para criar os IP's na interface localhost da máquina e que posteriormente são utilizados pelos sockets durante o bind.

A implementação deste trabalho foi baseada em quatro threads principais:

- Interface de linha de comando (CLI): Esta thread é responsável por tratar a entrada do usuário e designar o processamento dos comandos para os métodos adequados. Os comandos tratados são:
  - `add <IP>`: Adiciona um enlace entre o roteador atual e o roteador cujo IP da interface é especificado no parâmetro.
  - `del <IP>`: Remove o enlace entre o roteador atual e o roteador cujo IP da interface é especificado no parâmetro.
  - `trace` <IP>: Executa o comando trace, que traça uma rota entre o roteador atual e o destino, onde os roteadores intermediários adicionam seu endereço e encaminham para o próximo da rota. Ao chegar no destino final, o roteador que recebeu o trace envia uma resposta para o solicitante com o JSON contendo os caminhos intermediários.
  - `quit`: Encerra o programa.
- Recebimento de mensagens: Thread responsável por receber dados enviados ao roteador. Depois de feito o bind, essa thread usa o socket com um buffer de 1024 bytes e aguarda a chegada dos dados.
- Envio de mensagens de update: Thread responsável por realizar o envio de mensagens de atualização aos roteadores vizinhos contendo o peso das rotas adjacentes. O tempo entre cada atualização é especificado como parâmetro ao invocar o programa, conforme descrito anteriormente. Basicamente, esta thread executa um loop infinito no qual há uma chamada

ao método `sleep`, pelo tempo determinado na execução. Enquanto o programa estiver rodando, a cada intervalo de tempo uma atualização é enviada para os vizinhos.

- Verificação do tempo de vida das rotas: Esta thread é responsável por executar a cada 4 vezes o intervalo de tempo definido na linha de comando uma rotina que irá verificar as rotas desatualizadas no roteador. Caso a rota não receba uma mensagem de atualização de um dos vizinhos por um tempo superior a 4 vezes o determinado, a rota é removida.

As mensagens trocadas entre os roteadores utilizam codificação JSON dos dados, As mensagens trafegadas tem pelo menos três campos:

- source : Especifica o endereço IP do programa que originou a mensagem
- destination: Especifica o endereço IP do programa destinatário
- type: Especifica o tipo da mensagem enviada, que pode ser um dos listados a seguir:
    - update: Mensagem que carrega as informações de atualização das rotas.
    - data: Mensagem que carrega um dado qualquer.
    - trace: Realiza a função de rastreamento de rota.

As principais funcionalidades implementadas no trabalho estão descritas nas seções a seguir. Os exemplos utilizados nesta documentação se baseiam na topologia de exemplo existente no final do documento.

1. Atualizações Periódicas

As atualizações periódicas são enviadas em um intervalo de tempo determinado na chamada do programa pelo usuário. A thread responsável por essa função executa um loop que perdura durante a execução do programa e periodicamente envia uma mensagem JSON no seguinte formato:

```
{
'destination': '127.0.1.3',
'source': '127.0.1.1',
'type': 'update',
'distances':
      {
      '127.0.1.2': 2,
      '127.0.1.1': 0,
      '127.0.1.4': 4
      }
}
```

A seguir um trecho do código que mostra como essa funcionalidade foi implementada:

```
while(self.running):
       for neighbour in self.links.keys():
      UDP_MESSAGE['distances'] = {}
      UDP_MESSAGE['destination'] = neighbour
      for key, value in self.routingTable.items():
             if(key != neighbour):
                    UDP_MESSAGE['distances'][key] = value['weight']
      UDP_MESSAGE['distances'][self.addr] = 0
       self.sock.sendto(json.dumps(UDP_MESSAGE).encode(),(neighbour,self.UDP_PORT))
```

```
        time.sleep(int(self.period))
```

Este exemplo mostra uma mensagem de atualização proveniente do roteador 127.0.1.1, enviada ao seu vizinho 127.0.1.3 informando as distâncias dos seus enlaces. Esta mensagem é montada tomando como base a tabela de roteamento do roteador.

A tabela de roteamento é uma estrutura de dados do tipo dicionário e cada roteador a atualiza com base nos updates recebidos de seus vizinhos. A seguir um exemplo dessa estrutura:

```
{
'127.0.1.2':
      {
      'hops': [
            ['127.0.1.2', 4]
      ],
      'weight': 2,
      'nextHop': 0
      },
'127.0.1.4':
      {'hops': [
            ['127.0.1.4', 4],
            ['127.0.1.2', 4]
      ],
      'weight': 4,
      'nextHop': 0
      },
'127.0.1.3':
      {'hops': [
            ['127.0.1.3', 4],
            ['127.0.1.4', 4],
            ['127.0.1.2', 4]
      ],
      'weight': 5,
      'nextHop': 0
      }
}
```

Cada chave do dicionário acima corresponde à um destino da rede na qual o roteador está conectado. Para cada destino existem os seguintes campos:

- `hops`: Lista de próximos passos que o pacote pode tomar para ser encaminhado para o destino final. Cada elemento desta lista é outra lista que contém o endereço do roteador correspondente ao passo e um inteiro que guarda o tempo de vida dessa rota. Como será dito mais frente nessa documentação, cada rota tem um tempo de vida máximo que é renovado quando uma atualização correspondente à ela é recebida.
- `weigth`: Este campo corresponde ao peso da(s) rota(s) desta entrada na tabela. No exemplo acima, a tabela de roteamento diz que o host 172.0.1.4 pode ser alcançado com peso 4 através dos hosts `127.0.1.4` e `172.0.1.2`.
- `nextHop`: Este campo é utilizado para efetuar o balanceamento de carga entre as rotas quando mais de uma rota com o mesmo peso está disponível para um determinado pacote.

Pode ser entendido como um valor de TTL (*Time to Live*) O valor inicial é 4. Toda vez que uma atualização de rota chega ao roteador, a entrada correspondente na tabela tem seu valor atualizado para 4. Uma outra thread é responsável por por decrementar o valor de de todas os TTL's periodicamente. Se este valor chega a zero, a rota é removida.

2. Split Horizon

Para evitar o problema de contagem ao infinito, que pode ocorrer no roteamento por vetor de distância, foi implementada a otimização split horizon. As informações de update enviadas para um vizinho não contemplam atualizações de rota de e para este nó. Nó trecho de código mostrado acima, um dos condicionais faz essa verificação; Ao iterar sobre a tabela de roteamento, as informações de rota para o vizinho são ignoradas ao montar a mensagem de atualização para o mesmo.

3. Balanceamento de carga

A funcionalidade de balanceamento de carga permite que um pacote seja encaminhado para múltiplas rotas que tenham o mesmo peso para chegar ao destino final. Para exemplificar o funcionamento da implementação, vamos observar uma parte da tabela de roteamento:

```
'127.0.1.3':
      {'hops': [
            ['127.0.1.3', 4],
            ['127.0.1.4', 4],
            ['127.0.1.2', 4]
      ],
      'weight': 5,
      'nextHop': 0
}
```

O trecho do dicionário acima foi retirado da tabela de roteamento mostrada anteriormente. Ele mostra os caminhos disponíveis para chegar ao host `127.0.1.3`. Neste caso existem 3 rotas disponíveis, cada uma com o mesmo peso (5). O balanceamento de carga é feito utilizando a variável `hextHop`. Esta variável guarda o valor do índice de elemento da lista hops. No estado atual da tabela mostrada, o pacote que chegar no roteador em questão e que tiver como destino o host `127.0.1.3` utilizará a primeira rota da lista `hops`. Neste caso, o pacote sairá pelo roteador `127.0.1.3` (neste caso, pelo enlace conectado ao próprio destino). Depois desse encaminhamento, o valor de nextHop é incrementado em 1, de tal forma que o próximo pacote que chegar neste roteador e tiver o mesmo destino, utilizará a rota de índice 1, ou seja, o roteador `127.0.1.4`. Este processo ocorre sucessivamente até chegar ao final da lista. Depois de utilizar a última rota, o contador nextHop é zerado para distribuir o tráfego entre as rotas disponíveis novamente. O trecho de código a seguir mostra o método utilizado para fazer o balanceamento de carga:

```
def _nextHop(self, destination):
	nextHops = self.routingTable[destination]['hops']
	if(len(nextHops) > 1): # Load balancing
		nextHop = nextHops[self.routingTable[destination]['nextHop']]
		self.routingTable[destination]['nextHop'] += 1
		if(self.routingTable[destination]['nextHop'] > len(nextHops)-1):
		self.routingTable[destination]['nextHop'] = 0 # Volta para a primeira
rota
		return nextHop[0]

	else:
		return nextHops[0][0]
```

4. Reroteamento imediato

O reroteamento imediato é uma função utilizada quando há a remoção de um enlace que comprometa uma rota em utilização. Neste trabalho, a função foi implementada chamando a função de update ao remover um enlace. Quando ocorre uma remoção pelo comando del na linha de comando do programa, os roteador envia uma mensagem de update aos seus vizinhos refletindo essa mudança. O trecho de código a seguir ilustra como é feito o processo de remoção de enlace, em destaque para a chamada da rotina de envio de atualização aos vizinhos:

```
	def removeLink(self, destinationAddr):
		del self.links[destinationAddr]
		print (self.routingTable)
		for hop in list(self.routingTable.get(destinationAddr)['hops']):
			if(hop[0] == destinationAddr):
			self.routingTable.get(destinationAddr)['hops'].remove(hop)

		if(len(self.routingTable.get(destinationAddr)['hops']) == 0):
			self.routingTable.get(destinationAddr)['weight']= math.inf

		self.sendUpdate()
```

5. Remoção de rotas desatualizadas

A remoção de rotas desatualizadas é realizada a partir de um contador presente em cada rota. Para mostrar como foi feita a implementação, vamos observar novamente um trecho da tabela de roteamento:

```
'127.0.1.4':
	{'hops': [
		['127.0.1.4', 1],
		['127.0.1.2', 4]
	],
	'weight': 4,
	'nextHop': 0
	}
```

Este trecho da tabela mostra que existem duas rotas para o destino 127.0.1.4. A primeira é pelo roteador 127.0.1.4 e a segunda pelo 127.0.1.2. Essas informações estão presentes na lista hops. Cada elemento dessa lista contém o endereço de destino do próximo passo e um valor de TTL, que inicialmente vale 4. Uma outra thread é responsável por iterar todas as rotas e decrementar o valor de TTL de todas elas. Caso chegue uma mensagem de update correspondente à essa rota, o valor é setado para 4 novamente. Se a mensagem de update não chegar, o valor de TTL é decrementado até chegar em 0. Nessa situação, a rota em questão é removida. No exemplo acima, a rota passando pelo 127.0.1.4 está prestes a ser removida. Na próxima iteração da thread que verifica o tempo de vida das rotas este valor será decrementado para 0 se uma mensagem de update dessa rota não chegar. Se isso ocorrer, a rota será removida. Quando uma rota é removida, é verificado se para o destino em questão existe outra rota. Se for este o caso, a rota expirada é excluída e o programa continua normalmente. Caso a rota removida seja a única para aquele destino, ela é removida da mesma forma e o valor de weigth é setado para infinito, para sinalizar que aquele host está inalcançável devido a ausências de rotas.

O trecho de código a seguir mostra a verificação do tempo de vida das rotas:

```
def _tmoThread(self):

while(self.running):
      time.sleep(int(self.period)*4)
      with self.lock:
      for key, value in self.routingTable.items():
            for item in list(value['hops']):
                  ttl = int(item[1])
                  item[1] = ttl - 1

                  if(item[1] == 0): # Timeout na rota
                  value['hops'].remove(item)
                  value['weight'] = math.inf
                  if(item[0] in self.links):
                        del self.links[item[0]]
```

Topologia de exemplo: