Universidade Federal de Minas Gerais – Belo Horizonte, MG – 2019/1
Instituto de Ciências Exatas – Departamento de Ciência da Computação

DCC023: Redes de Computadores
TP1 – DCCRIP: Protocolo de Roteamento por Vetor de Distância

Aluna: Scarlet Gianasi Viana (2016006891)

1. Introdução

Este trabalho teve como objetivo implementar um roteador, **DCCRIP**, que utiliza o protocolo de vetor de distância para calcular rotas para o envio de mensagens com o menor custo.

2. DCCRIP

O protocolo de roteamento **DCCRIP**, como especificado, possui roteadores na interface de *loopback*, cada um com seu IP, na subrede *127.0.1.0/16*, com troca de dados a partir de um soquete UDP na porta *55155*. Cada roteador será conectado a outros roteadores a partir de um enlace com um custo determinado. Os enlaces formam a topologia da rede, e determinam os caminhos possíveis de envios de mensagens. Roteadores podem enviar mensagens de dados, requisição de tabelas de roteamento, requisição de payloads e updates para seus vizinhos.

2.1 Vetor de Distância

No protocolo por vetor de distância, cada roteador é primeiramente conectado a vizinhos: roteadores que possuem enlaces diretos com o mesmo. Periodicamente, cada roteador envia mensagens de update que indicam as menores rotas para cada vizinho que possui. Dessa forma, seus vizinhos podem descobrir outras rotas, através dos enlaces vizinhos, para outros roteadores que não estão conectados diretamente ao roteador.

2.2 Mensagens

As mensagens trocadas pelos roteadores seguem o formato **JSON**. Existe quatro tipos de mensagens, descritas a seguir:

2.2.1 Mensagem de Dados

As **mensagens de dados** serão enviadas quando um roteador a que foi enviada como destino receber uma mensagem de **Rasteamento (Trace)** ou de **Requisição de Tabela**, e o campo "**payload**" será diferente para cada caso. Se o roteador receber anteriormente uma mensagem **Trace**, "**payload**" conterá os "**hops**" da mensagem **Trace** recebida. Se a mensagem anterior for **Requisição de Tabela**, "**payload**" será uma string contendo tuplas no formato *`(destino, próximo hop, custo)`*, ordenadas seguindo o destino e o próximo hop.

O formato da mensagem de dados é:

```
{
   "type": "data",
   "source": "IP1",
   "destination": "IP2",
   "payload": hops [IP1, IP3, IP2] ou tuplas "[(destino, próx.Hop, custo), ...]"
}
```

2.2.2 Mensagem de Update

Cada roteador envia **mensagens de update** periodicamente para seus vizinhos. Cada mensagem possui o campo "**distances**", que contém a menor distância para cada destino conhecido de sua tabela de roteamento, exceto as rotas para o próprio vizinho, e rotas aprendidas do vizinho, evitando o problema da contagem ao infinito. Esta otimização se chama **Split Horizon**. Estas mensagens possibilitam o aprendizado de novas rotas pelos roteadores, assim como a

detecção de enlaces perdidos (a partir da remoção de rotas desatualizadas, apenas possível se houver constante atualização das mesmas), tornando possível o conhecimento de todas as rotas alcançáveis pelo roteador.

O formato da mensagem de update é:

```
{
  "type": "update",
  "source": "IP1",
  "destination": "IP2",
  "distances": {
    "IPX": C1,
    "IPY": C2,
    "IPZ": C3
  }
}
```

2.2.3 Mensagem de Rastreamento (Trace)

As **mensagens de rastreamento** são enviadas quando o comando "*`trace IP`*" é dado para algum roteador. O roteador enviará essa mensagem com o destino "**IP**", com o campo "**hops**" inicialmente apenas com seu próprio endereço. A cada roteador que receber a mensagem e repassá-la para o próximo hop em direção ao destino, mais um endereço é adicionado à lista "**hops**". Quando a mensagem chegar no destinatário, este campo conterá todos os roteadores que formaram o caminho da fonte ao destino.

O formato da mensagem de rastreamento é:

```
{
  "type": "trace",
  "source": "IP1",
  "destination": "IP2",
  "hops": [IP1, IPX, IPY, IPZ, IP2]
}
```

2.2.4 Mensagem de Requisição de Tabela

As mensagens de **requisição de tabela** são enviadas a partir de um roteador que recebeu o comando "*`table IP`*", e são enviadas para o endereço dado, para que o roteador que a enviou recebe uma mensagem de dados com o "**payload**" contendo os dados de sua tabela de roteamento, como previamente especificado.

O formato da mensagem de requisição de tabela é:

```
{
  "type": "table",
  "source": "IP1",
  "destination": "IP2"
}
```

2.3 Comandos

Os comandos aceitos por cada roteador são:

- **`add <IP> <custo>`**
  - Adiciona um enlace vizinho ao roteador no endereço `<IP>` com peso `<custo>`.
- **`del <IP>`**
  - Remove um enlace vizinho ao roteador no endereço `<IP>`.
- **`trace <IP>`**
  - Envia uma mensagem de rastreamento para `<IP>`.
- **`table <IP>`**
  - Envia uma mensagem de requisição de tabela para `<IP>`.
- **`quit`**
  - Finaliza a execução do programa.

2.4 Funcionamento geral do DCCRIP

Após a inicialização da topologia dos roteadores, os mesmos enviarão mensagens de update a cada **<Período> segundos** para seus vizinhos, que, ao receber cada mensagem, irá atualizar sua tabela de roteamento respectivamente aos dados recebidos. Enlaces podem ser removidos a qualquer momento, mas como cada roteador possui todas as rotas conhecidas e atualizadas, o reroteamento imediato (se existir um caminho até o destino) pode ser realizado.

Na tabela de roteamento de cada roteador, cada rota possui um estado que indica se foi atualizada ou não. Assim, a cada adição de um enlace ou atualização de rota, é possível saber quais rotas remover a cada **4*<Período> segundos**. Cada roteador pode receber os comandos descritos em **2.3**, e enviar mensagens respectivamente aos endereços dados, se existir um caminho do roteador até o IP. Caso não haja nenhuma rota para o endereço dado, a mensagem é descartada.

Analogamente, cada roteador recebe mensagens entre a fonte e o destino, e, se o mesmo não souber uma rota para o destino, a mensagem é descartada; porém isso deve acontecer principalmente nos casos em que realmente não existem mais rotas devido à remoção de enlaces, afinal as rotas são atualizadas com frequência.

3. Implementação

A implementação deste trabalho foi feita utilizando a linguagem **Python** (versão 3.6.7). O roteador recebe os argumentos de entrada, para cada roteador, seguindo o modelo especificado:

`./router.py <IP> <PERÍODO> [TOPOLOGIA]`

Em que **<Período>** corresponde aos segundos de envio de mensagens de update (e, consequentemente, aos 4*<Período> segundos de atualização de rotas desatualizadas), e **[TOPOLOGIA]** é um campo opcional que corresponde a um arquivo com comando de adição de enlaces que formam a topologia inicial da rede.

3.1 Tabela de Roteamento

A **tabela de roteamento** do roteador, como é por **vetor de distância,** possui, para cada endereço-destino, endereços que são os próximos *hops* a se enviar uma mensagem para se chegar no destino, com o custo total da rota respectivo. Para cada destino, é possível existir diversas rotas conhecidas, para que, mesmo se uma rota for perdida devido à remoção de um enlace, o roteador consiga enviar mensagens utilizando outra rota que não o utilize, se existir.

Na implementação do código, a tabela de roteamento é um dicionário, cujas chaves são os endereços-destino, e cujos valores são listas de dicionários, com os campos "**nxtHop**", "**weight**" e "**update**". "**nxtHop**" é o endereço de um vizinho do roteador que possui uma rota ao destino ou o endereço do próprio roteador (cujo destino é ele mesmo e possui custo 0). "**weight**" é o custo total até o endereço-destino, e "**update**" é um booleano que indica se a rota está atualizada ou não (*True* se sim, *False* se não).

Uma tabela de roteamento seguem o seguinte formato:

```
table = { IP1: [{'nxtHop': IP1, 'weight': 5, 'update': True}, … ],
          IP2: [{'nxtHop': IP3, 'weight': 3, 'update': True}],
          …
          IPX = [],
          …
        }
```

Dessa forma, quando um roteador deseja enviar uma mensagem para um endereço, basta o mesmo buscar uma das rotas cuja chave é o destino, e encontrar a menor rota. O próximo hop então será dado por "**nxtHop**", e a mensagem será enviada para o endereço encontrado.

Com o campo "**update**", a cada atualização de rota, as rotas atualizadas receberão *True*, e a cada remoção de rotas desatualizadas todas as rotas com *False* poderão ser removidas. Este processo será explicado com mais detalhes na seção **3.4.4**.

3.2 Atualização de Rotas

Cada vez que um roteador receber uma mensagem de update de um vizinho seu, ele checará o campo de distâncias da mensagem. A partir das informações dadas, ele checará os novos destinos recebidos, e atualizará sua tabela de roteamento com as novas rotas encontradas.

Por exemplo, se um roteador **A** recebe uma mensagem de update do roteador **B**, ele sabe que as distâncias para cada destino (suponha, destinos **C** e **D**) serão dadas pelo peso de **A** → **B** mais o peso de **B** → **C** ou **B** → **D**. Portanto a tabela de A possuirá mais dois novos destinos, **C** e **D**, com próximo hop dado por **B**.

Se as rotas pelo mesmo hop já existem, o único campo que muda é "**update**", que receberá *True*.

Pode ser que um enlace de um vizinho de **B** que não é vizinho de **A** foi removido, então isso poderá ser tratado depois, já que cada rota de **B** que não foi atualizada em 4***<Período>** segundos recebe *False* em "**update**". Dessa forma, o roteador pode manter rotas alternativas para destinos, que, quando forem removidas, ainda existam outras rotas para enviar mensagens.

3.3 Troca de mensagens

Para realizar a troca de mensagens, além de receber comandos do usuário, foram criadas threads para cada processo. Uma thread é responsável por ler input do usuário, e chamar funções quando necessário. Outra thread é responsável pelo recebimento de mensagens, com seu respectivo tratamento.

Cada mensagem recebida cujo destinatário não for o próprio roteador é enviada para o próximo vizinho cujo custo da rota para o destino é o menor possível, se existir uma rota. Se não, a mensagem é descartada. Se a mensagem é para o roteador, ele a trata-rá de acordo com o que foi especificado nas seções anteriores, enviando novas mensagens ou imprimindo resultados na tela.

Outras duas threads para temporização foram criadas. Uma delas executa uma função que envia mensagens de update para os vizinhos do roteador a cada **<Período>** segundos. A outra remove rotas desatualizadas a cada 4***<Período>** segundos. Mantendo a sincronia de cada operação, o roteador exerce sua função de troca de mensagens e atualização de rotas.

3.4 Funcionalidades

Foram especificadas algumas otimizações a serem implementadas no roteador. Algumas já foram brevemente descritas na documentação, mas a seguir está uma descrição mais completa das soluções encontradas para a implementação.

3.4.1 Atualizações periódicas

Para realizar atualizações periódicas, foi utilizado um temporizador com a biblioteca "**threading**", que chama a função "**update**()", que por si chama a função "**sendUpdateMsg**()", que envia uma mensagem de update para todos os seus vizinhos, a cada "**pi**" (**<Período>**) segundos. Como abaixo:

```
def update():
    sendUpdateMsg()
    Timer(pi, update).start()
```

A função é inicialmente chamada logo após a inicialização do programa, e início das threads principais.

3.4.2 Split Horizon

Para implementar o **Split Horizon** e evitar o problema da contagem para o infinito, quando é criada a lista de distâncias para ser enviada na mensagem de update para o vizinho **A**, duas condições são testadas: para cada roteador destino na tabela, o roteador **A** é ignorado, e para cada rota, se o próximo hop é **A**, a rota também é ignorada.

No código abaixo, que calcula a lista de distâncias, as condições descritas estão em negrito:

```
def createDistancesList(dest):
    dl = {}
    minWeight = 10000
    for router in dt:
        if router != dest: # Split horizon
            if dt[router]:
                for route in dt[router]:
                    if route['nxtHop'] != dest: # Split horizon
                        if route['weight'] < minWeight:
                            minWeight = route['weight']
                if minWeight < 10000:
                    dl[router] = minWeight
                minWeight = 10000
    return dl
```

3.4.3 Balanceamento de Carga

O balanceamento de carga consiste em balancear o fluxo das rotas de forma que, por exemplo, se houverem duas rotas com o mesmo custo, o roteador utilize 50% do fluxo em cada rota. Para isso, a cada update de rotas, é checado se existem rotas com custo igual para o mesmo destino. Assim, o roteador alternará o envio de mensagens para as rotas com custo igual, de forma evitar o congestionamento do tráfego nos enlaces.

Para isso, a implementação foi feita utilizando um dicionário auxiliar, uma **tabela de balanceamento**, que, após cada update ou adição de enlaces, serão checadas rotas para o mesmo destino com o mesmo custo, e estas serão inseridas na tabela na chave “destino”, guardando apenas os endereços dos hops.

Toda vez que se desejar enviar uma mensagem, e o melhor custo para tal corresponder ao custo de rotas empatadas, o roteador *sorteará randômicamente* um número que corresponde ao índice da rota na tabela de balanceamento, e então a rota será escolhida. Com um númer suficiente de envios de mensagens de rotas empatadas, estatisticamente terão uma proporção equilibrada de envio para cada rota, garantindo o balanceamento. Este método também é eficiente para que vizinhos com rotas empatadas que utilizem o mesmo método de balanceamento tenham mais chances de enviar rotas diferentes entre si, afinal, a escolha é sempre randômica.

Quando um enlace for removido, e seu endereço for uma chave na tabela de balanceamento, essa chave também será removida da tabela. Quando uma rota for removida por estar desatualizada, se ela for uma das rotas com custo igual, a chave de seu destino na tabela de balanceamento também será removida.

A função abaixo é chamada toda vez que uma mensagem é enviada para um destino cujo menor custo contêm rotas empatadas, sorteia um inteiro de acordo com o tamanho da lista de próximos hops para que ele seja o destino da próxima mensagem:

```
def loadBalancing(dest):
    index = random.randint(0,len(loadBalance[dest])-1)
    hop = loadBalance[dest][index]
    return hop
```

A função abaixo é chamada toda vez que um enlace ou uma rota for adicionado, encontra rotas na tabela de roteamente que possuem o mesmo custo para o mesmo destino:

```
def checkRoutes(router, nextHop, weight):
    for route in dt[router]:
        if (route['nxtHop'] != nextHop) and route['weight'] == weight:
            addRouteToLoadBalance(router, route['nxtHop'], nextHop)
```

Finalmente, a função abaixo é chamada toda vez que alguma rota ou enlace for removido, e retira as rotas empatadas respectivas. Se apenas sobrar uma rota na lista, não há mais empate, e então o destino é retirado da lista de balanceamento:

```
def checkBalanceTable(key, address):
    if address in loadBalance[key]:
        loadBalance[key].remove(address)
        if len(loadBalance[key]) < 2:
            loadBalance.pop(key, None)
```

3.4.4 Reroteamento Imediato

Para o reroteamento imediado, na tabela de roteamento são armazenadas mais de uma rota para o mesmo destino, contanto que não sejam do mesmo vizinho (pois isso significa que são rotas iguais, provavelmente de updates seguidos) portanto mesmo que a rota de menor custo seja removida por alguma razão (por exemplo, remoção de um enlace), o destino continuará alcançável se existir alguma outra rota para o mesmo.

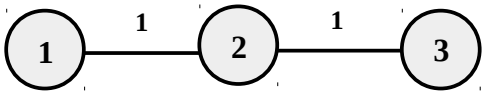
3.4.5 Remoção de Rotas Desatualizadas

Para a remoção de rotas desatualizadas, assim como nas atualizaçõs periódicas, foi implementado um temporizador, dessa vez com tempo 4***<Período>** segundos, que chama uma função que percorre a tabela de roteamento buscando rotas com "**update**" = *False*, que são desatualizadas e, portanto, removidas. Para isto funcionar, toda vez que o roteador recebe uma mensagem de update, as rotas recebidas recebem atualizações em "**update**".

```
def updateRoutes():
    removeOldRoutes()
    Timer(4*pi, updateRoutes).start()
```

4. Testes

Testes foram realizados com diferentes topologias, mas como cada roteador utiliza um terminal, para evitar que a documentação se extenda além do necessário, algumas mensagens de logging demonstram o funcionamento de um roteador com a seguinte topologia inicial:



Em que cada nó é um roteador com endereço 127.0.1.#, com # de 1 a 3, respectivamente. O resultado do roteamento está em anexo, nos arquivos "**logging<IP>**" de cada endereço.

5. Conclusão

A partir da implementação do roteador DCCRIP, foi possível colocar em prática o que foi aprendido sobre a camada de rede, principalmente o protocolo de vetor de distância. O roteamento é essencial para que os dados cheguem a seus destinos, e a escolha de menores caminhos, mesmo com a possibilidade de perda de enlaces a qualquer instante, é um problema que exige constante atenção e atualização de dados sobre as rotas.