Redes de Computadores 2018/1
Trabalho Prático 2

Fábio Dayrell F. M. Rosa[1]

[1]Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Belo Horizonte – Minas Gerais – Brasil

dayrell@dcc.ufmg.br

1. Introdução

Nesta documentação está descrita a solução para o trabalho prático 2 da disciplina Redes de Computadores da UFMG. Inicialmente, é discutido o problema. Em seguida, será apresentado o desenvolvimento da solução apresentada para o problema, bem como as dificuldades e problemas encontrados durante a execução.

2. O problema

O problema proposto neste trabalho é o desenvolvimento de um emulador de um roteador que utiliza roteamento por vetor de distância. Este roteador deve implementar balenceamento de carga, reroteamento imediato, atualizações periódicas, dentre outras funcionalidades que serão discutidas nesta documentação.

3. Informações técnicas

O trabalho foi desenvolvido utilizando Python 3.6.5 no sistema operacional Mac OSX. Além disso, o programa foi testado em uma máquina virtual na Amazon com o sistema operacional Ubuntu 16.04.2 LTS para atestar que ele é compatível com o Linux.

Para executar o emulador do roteador, basta executar o comando abaixo:

```
python3 router.py <ADDR> <PERIOD> [STARTUP]
```

O valor [STARTUP] é opcional e trata-se de um arquivo de texto com as conexões de um roteador.

Após o início da sua execução, o servidor irá imprimir no terminal o IP da máquina em que ele está funcionando.

Além disso, é possível executar o roteador seguindo outro padrão, como mostrado abaixo:

```
python3 router.py --addr <ADDR> --update-period <PERIOD>
                              --startup-commands [STARTUP]
```

Sendo o valor [STARTUP] mais uma vez opcional.

4. Solução

Após a definição dos parâmetros de entrada e, caso não haja nenhum erro, a função $inicia\_roteador$ é inicializada. Nessa função, a conexão UDP é inicializada para que o roteador possa se comunicar com os demais. Também, é inicializado o objeto $roteador$ da classe $Roteador$, que traz consigo métodos relacionados à comunicação entre os roteadores.

Esse objeto é inicializado da seguinte forma:

```
def __init__(self, ip, period, server_socket):
    self.ip = ip
    self.roteadores = {self.ip: 0}
    self.period = int(period)
    self.vetor_distancia = {ip: {ip: 0}}
    self.socket = server_socket
    self.fila_proximo_roteador = {}
    self.ultimo_update = {}
```

O dicionário $roteadores$ é onde ficam armazenados os roteadores vizinhos ao roteador em execução. Já o dicionário de dicionários $vetor\_distancia$ armazena todas as possíveis distâncias até cada um dos roteadores. O trecho abaixo exemplifica o seu uso:

```
{
 ...
 127.0.1.2: {127.0.1.3: 1, 127.0.1.4: 2}
 ...
}
```

Com isso, é possível saber que para chegar até o roteador 127.0.1.2 é possível enviar uma mensagem para 127.0.1.3 com custo 1 e para 127.0.1.4 com custo 2.

O dicionario $fila\_proximo\_roteador$ guarda os roteadores com o menor custo para um destino desejado. Cada chave desse dicionário é um roteador alcansável e valor dessa chave são os roteadores com menor custo. O uso desse dicionário será discutido na subseção 4.2.

O dicionário $ultimo\_update$ guarda o momento da última atualização recebida de cada roteador conhecido. O uso desse dicionário será discutido na subseção 4.4.

Além disso, foi feita a opção de criar três threads diferentes para a execução do roteador. A primeira responsável por enviar a atualização de rotas para roteadores vizinhos, a segunda para a recepção de mensagens de outros roteadores e a terceira para ler o input do usuário, que pode adicionar ou remover novos links a qualquer momento.

A thread de recepção de mensagens inicializa a função $recebe\_mensagens$, que simplesmente "escuta"o socket criado para o roteador. Ao receber alguma mensagem de outro roteador, a função $define\_tipo\_mensagem$ é chamada para definir se essa é uma mensagem de dados,, de update ou de trace.

A thread de input do usuário aguarda por comandos do usuário no terminal e encerra a execução quando o usuário pressiona ctrl+c ou envia $quit$.

Nas próximas subseções, serão discutidas as implementações de requisitos solicitados no enunciado do trabalho prático.

4.1. Atualizações Periódicas

O trabalho de enviar atualizações periódicas para os roteadores vizinhos é feito pela função $update\_rotas$, que é inicializada por uma thread, como dito anteriormente. Essa função simplesmente itera indefinidamente, sempre verificando se o tempo definido pelo usuário na inicialização do programa já foi excedido.

Para isso, foi utilizado o método $perf\_counter()$ do Python, como pode ser visto no trecho de código abaixo.

```
def update_rotas(self):
    tempo_inicio = time.perf_counter()
    period = self.period

    while True:
        tempo_atual = time.perf_counter()

        if (tempo_atual - tempo_inicio) > period:
            self.verifica_roteador_indisponivel()
            self.envia_atualizacao()
            tempo_inicio = time.perf_counter()
```

4.2. Balanceamento de Carga

Como foi definido no enunciado do trabalho prático, caso existam duas ou mais rotas com peso igual para um roteador e esse for o menor peso possível, é preciso que o tráfego seja direcionado de modo proporcional entre essas rotas.

Para implementar esse balanceamento, foi criada uma estrutura adicional que funciona como uma fila, a lista $fila\_proximo\_roteador$. Esse balanceamento é feito no método $proximo\_roteador$, que retorna a rota que será usada para enviar uma mensagem.

Todas as menores rotas para um roteador X ficam armazenadas nessa fila. A função $proximo\_roteador$ irá sempre retornar o primeiro elemento da $fila\_proximo\_roteador$, remover esse elemento e adicioná-lo ao fim dessa lista. Dessa forma, é garantido que, pelo menos estatisticamente, haverá balanceamento de carga.

4.3. Reroteamento Imediato

A decisão de qual rota será utilizada para enviar uma mensagem é tomada sempre no momento do envio através da função $proximo\_roteador$. Como todas as informações de distâncias do roteador até todos os outros roteadores ficam armazenadas no dicionário de dicionários $vetor\_distancia$, quando uma rota é removida, caso exista uma outra rota disponível, ela será escolhida nessa função.

4.4. Remoção de Rotas Desatualizadas

Além de enviar atualizações periódicas, o método descrito na subseção 4.1 também verifica roteadores inativos através do método $verifica\_roteador\_indisponivel$.

O dicionário $ultimo\_update$ armazena o momento em que cada roteador conhecido enviou o seu último update. Como dito no parágrafo anterior, sempre que o roteador em execução envia um update, a função $verifica\_roteador\_indisponivel$ é executada.

Essa função verifica para todos os roteadores conhecidos, se existe algum que está sem enviar um update há mais de três vezes o período informado pelo usuário na entrada do programa. Caso existe algum, esse roteador é removido da lista de roteadores conhecidos pelo roteador em execução.

4.5. Split Horizon

Para que um roteador em execução não envie para o roteador X a distância dele até X, bastou incluir a seguinte condição no método $menores\_distancias$:

```
if roteador != destination:
    novo_vetor_distancias[roteador] = menor_distancia
```

5. Dificuldades de implementação

Pela primeira vez em um trabalho prático dessa disciplina foi utilizado o uso de sockets do tipo UDP. O seu funcionamento é bastante diferente do TCP, utilizado nos últimos dois trabalhos. Por isso, foram necessários alguns testes iniciais para entender como utilizá-lo com o Python.

Além disso, a criação e parseamento de objetos do tipo json foi um desafio. No entanto, acredito que o uso desse tipo de objeto no contexto desse trabalho foi excelente, uma vez que facilita bastante a comunicação entre os roteadores. Também, não é difícil trabalhar com objetos json quando se está utilizando o Python.

6. Conclusões

Esta documentação descreveu, em alto nível, o problema proposto no trabalho prático 2 e, também, as soluções implementadas para que ele pudesse ser resolvido.

Com este trabalho foi possível implementar uma aplicação UDP. Além disso, foi possível conhecer e implementar políticas e métodos relacionados ao roteamento de pacotes em uma rede.

Além disso, é interessante implementar um emulador que utiliza um protocolo definido, de modo que o programa desenvolvido por mim é capaz de conversar com qualquer outro programa (emulador) implementado por outras pessoas e que utilizem o mesmo protocolo.