Trabalho Prático 2

DCCRIP: Protocolo de Roteamento por Vetor de Distância

Redes de Computadores - 2019/01
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

Deiziane Natani da Silva
2015121980

---

1. Introdução

O presente trabalho consiste em implementar o DCCRIP, um roteador que utiliza roteamento por vetor distância. O DCCRIP tem suporte a pesos nos enlaces, balanceamento de carga, medição de rotas e outras funcionalidades. A codificação foi feita em Python, e a biblioteca JSON foi utilizada para facilitar o envio de mensagens.

2. Desenvolvimento

O roteador foi implementado através de uma classe chamada Router. A classe possui o IP, porta de operação, período de envio da mensagem de atualização de rotas, tabela de roteamento e tabela de histórico de rotas. A tabela de histórico mantém todas as rotas recebidas e é fundamental para resolver uma das questões citadas abaixo, como o reroteamento imediato. A tabela de roteamento é construída sob demanda sempre que um método necessita da melhor rota.

```python
# pega a melhor opção de rotas para cada IP
# cada uma pode ter mais que uma rota com a mesma distancia (balanceamento de
carga)
def update_routing_table(self):
    routes_by_ip = dict()

    # extrai rotas da tabela de histórico
    for history in self.history_table:
        ip_key = history['ip']
        # caso não haja uma rota para o IP na tabela de roteamento
        # ou a tabela de histórico tenha uma rota com distância menor, atualiza a
        entrada da tabela de roteamento
        if ip_key not in routes_by_ip.keys() or routes_by_ip[ip_key][0]
        ['distance'] > history['distance']:
            routes_by_ip[ip_key] = [history]
        elif routes_by_ip[ip_key][0]['distance'] == history['distance']:
            # caso haja uma rota para o IP com a mesma distancia, adiciona nova
            opção
            routes_by_ip[ip_key].append(history)

    self.routing_version = self.history_version
    self.routing_table = routes_by_ip
```

É importante ressaltar que rotas com mesmo custo também são adicionadas à tabela de roteamento, já que podem ser usadas no balanceamento de carga, discutido com mais detalhes no tópico 2.3.

Tanto a tabela de histórico quanto a tabela de roteamento possuem um número de versão. Ambas devem ter sempre a mesma numeração, caso sejam diferentes (caso em que a tabela de histórico é atualizada) é necessário construir novamente uma nova versão. Caso tenham o mesmo número, esse processo não é necessário.

```python
# atualiza tabela de roteamento por demanda
def get_routing_table(self):
    # verifica versão da tabela
    if self.routing_version < self.history_version:
        # atualiza a tabela de roteamento
        self.update_routing_table()
    return self.routing_table
```

2.1. Atualizações Periódicas e *Split Horizon*

No protocolo RIP (Routing Information Protocol) mensagens precisam ser trocadas periodicamente entre os roteadores, ou seja, a cada intervalo de tempo, mensagens de update são enviadas. O tempo é passado por parâmetro, e a função é implementada por meio de threads. Por esse motivo, o update pode acontecer paralelamente à outras atividades.

O Split Horizon consiste em um método de evitar o problema de contagem infinita. O método é uma técnica simples e eficaz de evitar loops em uma rede, ele faz com que um roteador não possa enviar nenhuma atualização sobre uma rota pela mesma interface que ele recebeu atualização sobre essa rota. A implementação é feita da seguinte forma: para cada vizinho, primeiro se exclui a rota que vai para ele mesmo e também as rotas recebidas por esse vizinho. Só assim a mensagem final é enviada.

A função de envio de mensagem de update tem a seguinte estrutura:

```
def send_update(self):
    routing_table = self.get_routing_table()

    update_message = dict()
    update_message['type'] = 'update'
    update_message['source'] = self.ip
    update_message['destination'] = ''
    update_message['distances'] = dict()

    for ip in routing_table.keys():
        update_message['distances'][ip] = routing_table[ip][0]['distance']

    connection = sck.socket(sck.AF_INET, sck.SOCK_DGRAM)
    for neighbor in self.neighbors_table:
        # copia mensagem para usar a original em outra vizinhança
        copy_message = copy.deepcopy(update_message)
        copy_message['destination'] = neighbor['ip']

        # split horizon, remove rota para o destino da mensagem
        if neighbor['ip'] in copy_message['distances'].keys():
            del copy_message['distances'][neighbor['ip']]

        # split horizon, remove rota aprendida do destino da mensagem
        to_remove = []
        for ip in routing_table.keys():
            learned_from_destination = list(filter(lambda option: option['next'] == neighbor
            ['ip'],
                                                   routing_table[ip]))
            if len(learned_from_destination) > 0 and ip in copy_message['distances'].keys():
                to_remove.append(ip)
        for ip in to_remove:
            copy_message['distances'].pop(ip)

        # todas as rotas tem a mesma porta
        connection.sendto(json.dumps(copy_message).encode(), (neighbor['ip'], self.port))
```

A função *send_update* é chamada através de um método auxiliar para disparar as mensagens de update depois de um tempo específico.

2.2. Balanceamento de Carga

A tabela de roteamento é utilizada para a implementação do balanceamento de carga. Como a tabela guarda rotas que possuem a mesma distância num vetor, como citado anteriormente, podemos usá-las para balancear a carga da rede. São usados números aleatórios de 0 ao tamanho do vetor, esse número corresponderá à rota que será utilizada para o envio da mensagem.

```
def send_message(self, message):
    routing_table = self.get_routing_table()

    if message['destination'] in routing_table.keys():
        # seleciona uma das melhores opções de rota (balanceamento de carga)
        options = routing_table[message['destination']]
        selected_option = randint(0, len(options) - 1)
        selected_hop = options[selected_option]['next']

        connection = sck.socket(sck.AF_INET, sck.SOCK_DGRAM)
        # todos os roteadores tem a mesma porta
        connection.sendto(json.dumps(message).encode(), (selected_hop, self.port))
```

2.3. Reroteamento Imediato

A tabela de histórico, também é fundamental para a implementação de reroteamento imediato. Caso haja uma mudança na rede (uma rota deixou de existir, por exemplo), também ocorre a mudança da versão da tabela de histórico. Como essa versão é diferente da versão da tabela de roteamento, essa é atualizada e uma nova rota é incluída no lugar da que deixou de existir, não causando assim problemas de envio.

2.4. Remoção de Redes Desatualizadas

Como no protocolo RIP é necessário o envio de updates regularmente pelos roteadores, caso um não envie mensagens em um tempo superior à 4π (com π = periodo de envio de updates) a rota deve ser removida da tabela de roteamento. Para isso, o algoritmo usa um TTL (*Time To Live*) para cada rota. O TTL das rotas é definido para 4, e a cada mensagem de update recebida que contém aquela rota, o número retorna para 4. Se nenhuma mensagem de update contém aquela rota, o número de TTL é decrementado em 1 unidade e caso chegue a 0 essa rota é removida.

```
def subtract_ttl(self, source_ip):
        to_remove = []
        # subtrai TTL das rotas aprendidas pela fonte
        for route in self.history_table:
            if route['next'] == source_ip:
                route['ttl'] = route['ttl'] - 1
                if route['ttl'] == 0:
                    to_remove.append(route)

        # remove rotas com TTL = 0 (rotas desatualizadas)
        for zero_ttl in to_remove:
            self.history_table.remove(zero_ttl)
```

3. Conclusão

O trabalho foi de grande importância no entendimentos do conhecimento adquiridos até a segunda parte da disciplina. Pudemos ver o funcionamento do que foi aprendido e enfrentar os desafios encontrados em implementá-los. Foi possível fixar melhor o conteúdo e compreendermos seus pormenores, que são de extrema importância no conhecimento de redes.