Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Redes de Computadores
Prof Dorgival

DCCRIP

Roteamento por Vetor de Distância

Luisa Bezerra
Tatiana Camelo

1. Introdução

Este projeto consiste na definição de um sistema de roteamento por vetor de distância com suporte a pesos dos enlaces, balanceamento de carga, medição de rotas e outras funcionalidades.

É criada uma rede de conexão dos roteadores via socket UDP para troca de e procura-se conhecer o menor caminho entre dois pontos, levando em consideração o peso das arestas no caminho entre o início e o destino. O programa é atualizado em tempo de execução, o que quer dizer que pontos roteadores podem ser adicionados ou retirados e assim, novos caminhos mínimos podem ser gerados, além da perda de comunicação entre dois pontos.

Para a execução do programa, usa-se o comando abaixo. Os parâmetros são <endereço de IP> <tempo de update>. O parâmetro [STARTUP] é opcional para adicionar vários comandos de criação da topografia da rede por meio de um arquivo de entrada.

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

A execução é interrompida quando houver sinal de interrupção como *ctrl-c* ou quando receber o comando quit do teclado.

2. Modelagem

2.1. Criação da topografia

Um sistema de roteamento por vetor de distância é um protocolo de redes de computadores para que cada roteador ache melhor distância conhecida até cada destino, e essa informação é mantida de forma dinâmica, já que novos IPs de acesso, que usam o porto *55151*, podem ser adicionados ou retirados durante a execução. A distância inferida no sistema é pela ordenação dos roteadores como nodos e suas ligações como vetores com pesos diferentes, que representa o tempo gasto devido a distância da comunicação.

A execução se dá baseada em dois conceitos para montagem da rede, em que se adiciona um novo roteador com peso ou deleta a ligação do roteador corrente.

```
add <ip> <weight>
```

```
del <ip>
```

As mensagens trocadas pelos roteadores são na forma de JSON objects e possuem pelo menos três campos para a comunicação: *source, destination, type*. Estas informações especificam o endereço de IP do remetente, endereço de IP do destinatário e o tipo da mensagem.

2.2. Rastreamento

Neste trabalho, as mensagens enviadas contemplam três tipos de informações: data, update e trace. Trace é usado para medir o tamanho mínimo entre dois roteadores e armazena um campo hops para armazenar uma lista de roteadores já visitados. Ao chegar no destino, o roteador envia à origem uma string em JSON indicando a rota feita.

Esta mensagem é enviada através do comando:

```
trace <ip>
```

A lógica de rastreamento é similar a Bellman-Ford. A mensagem de trace é enviada de um roteador destino para um roteador de origem – isso é, o roteador de origem envia uma mensagem ao destino e verifica o caminho entre eles. O roteador destino recebe esta mensagem e envia o trace, como se fosse uma mensagem de ACK, retornando o playload da mensagem original.

3. Implementação

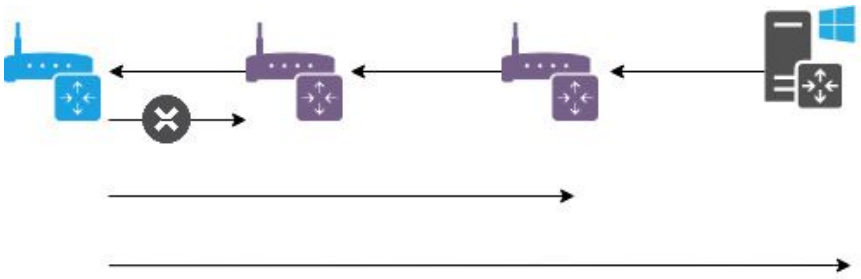
3.1. Atualizações Periódicas

É passado por parâmetro o tempo de update das rotas da topografia. Foi definido no sistema por sugestão do professor Ícaro que o tempo default de expiração fosse de quatro vezes o tempo de update. Esta é uma boa solução pois evita sobrecarga de verificação e também evita que ligações entre roteadores desligados ocupem muito espaço nas listas de caminhamento, sendo um número razoável para manter qualidade do controle.

```
TEMPO_ATUALIZACAO = tempoAtualizacao()
TEMPO_EXPIRACAO   = 4*TEMPO_ATUALIZACAO
```

3.2. Split Horizon

É um otimização para o envio de mensagens para evitar loop nas rotas. Consiste em enviar o pacote de dados e propagar para todos os nodos vizinhos, menos para o próprio roteador que atualizou sua tabela.



No código, isto é garantido através de uma conferência ao enviar mensagem de atualização para que não mande a mensagem para o endereço próprio, evitando assim o loop.

```
def processar_mensagem(self, mensagem):
    for roteador in self.tabela_distancias:
        if (roteador != self.ip_endereco):
            self.socket.sendto(mensagem.encode('utf-8'), (destino, self.port))
```

3.3. Balanceamento de Carga

O balanceamento de carga é a escolha aleatória entre caminhos de mesmo peso para desempate. No caso de mais de um caminho que tenha o mesmo custo, o sistema escolherá um e o tornará o caminho salvo na tabela.

```
    lista_roteadores_minimos = []
    for roteador_vizinho in self.tabela_distancias[roteador_destino].items()
    :
        if self.tabela_distancias[roteador_destino][roteador_vizinho][1] ==
        caminho_minimo:
            lista_roteadores_minimos.append(roteador_vizinho)

    roteador_escolhido = random.choice(lista_roteadores_minimos)
```

3.4. Reroteamento Imediato

Consiste na revalidação da menor rota caso algum nodo da então menor rota tenha sido deletado. A ideia é que sempre tenha atualizada a rota de menor custo. Na implementação do projeto, a função ***saltoDeRoteador*** faz a verificação da menor rota sempre que atualiza a tabela de distâncias.

```
atualizaTabelaDistancias(self.tabela_distancias)

if roteador_destino in self.tabela_distancias:
    caminho_minimo = min(dict_roteadores[1] for dict_roteadores in
    self.tabela_distancias[roteador_destino].values())
```

3.5. Remoção de Rotas Desatualizadas

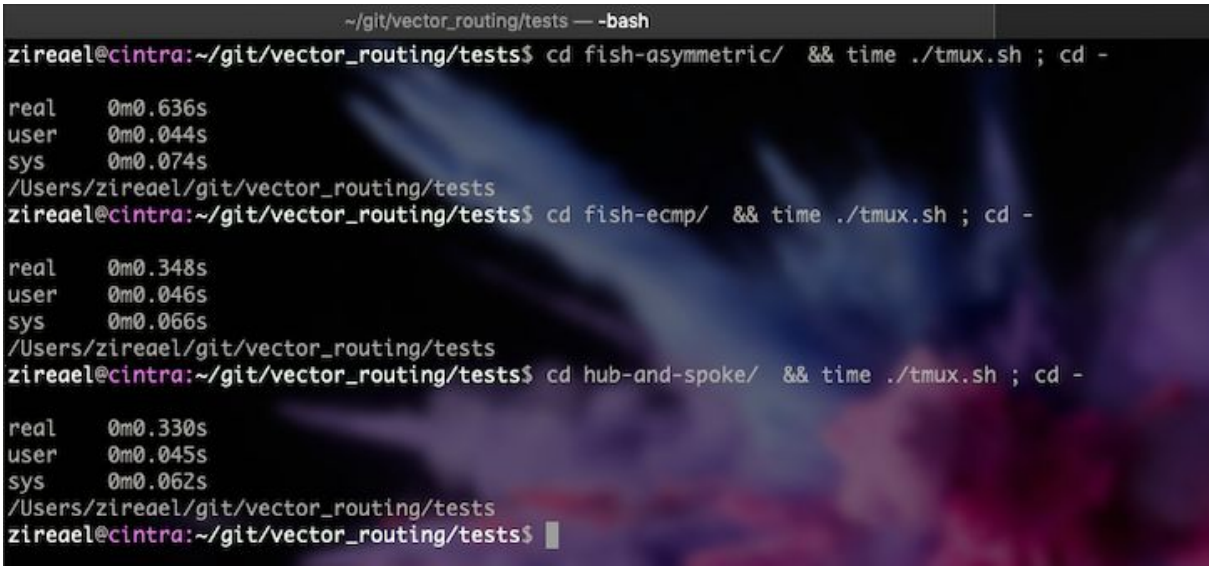
Ao receber atualização de ligações entre os gateways, é feita uma comparação com o timestamp da mensagem e do horário atual. Se a diferença entre eles for menor que o tempo de expiração, então considera-se que a informação ainda é válida e ela se mantém na lista, que é feito através da criação de uma lista auxiliar para replicar as rotas. Caso contrário, ela passa a ser ignorada e não entra na lista auxiliar e assim o sistema para de considerar aquele caminho como válido.

```
if time.time() - lista_peso_tempo[1] <= TEMPO_EXPIRACAO:
                copia_tabela_distancias[destino][roteador_vizinho] = lista_peso_tempo
```

4. Análise

Para a análise, foi escolhido como critério a velocidade de execução nos casos de teste apresentados pelo professor Ítalo. Mais sobre esta escolha estará discriminada na conclusão do projeto.

Foram três casos de teste apresentados, dois com topografia peixe, sendo uma simétrica e outra assimétrica. Foi interessante ver que a simetria mudou significativamente o tempo de execução dos scripts.



Exemplo de execução.

```
trace 127.0.1.3
O vizinho é 127.0.1.3
Processando mensagem, que é {'type': 'data', 'source': '127.0.1.3', 'destination': '127.0.1.1', 'payload': '{"type": "trace", "source": "127.0.1.1", "destin
ation": "127.0.1.3", "hops": ["127.0.1.1", "127.0.1.3"]}'}
Mensagem de trace  {'type': 'trace', 'source': '127.0.1.2', 'destination': '127.0.1.1', 'hops': ['127.0.1.2', '127.0.1.1']}
O vizinho é 127.0.1.3
```

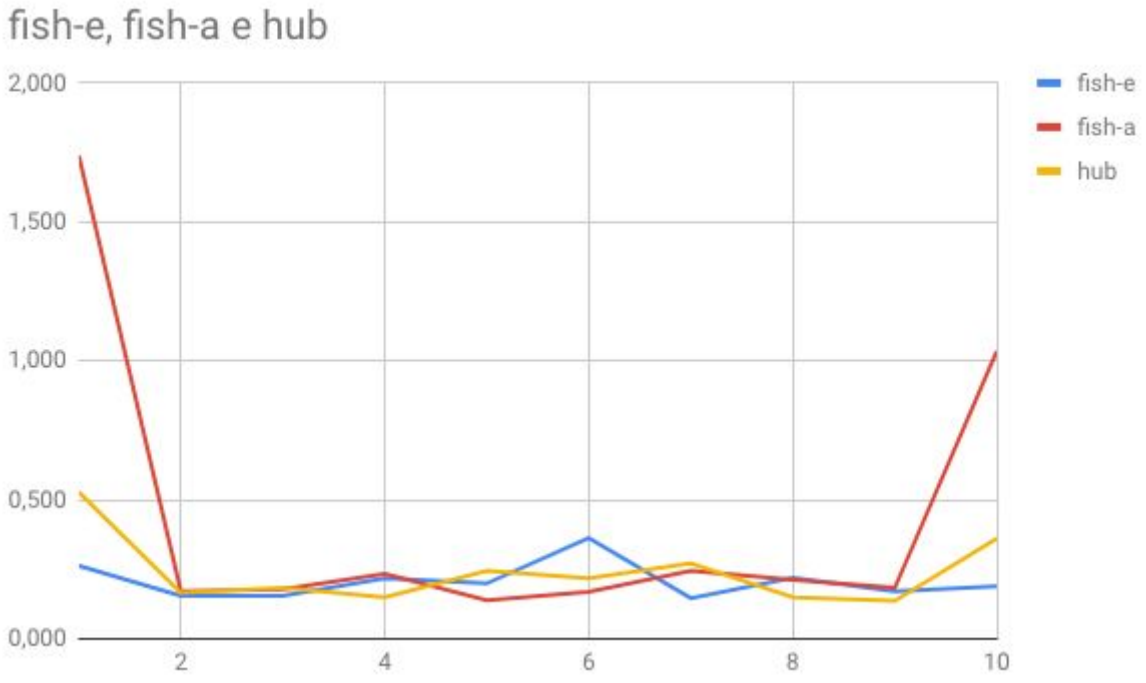
```
add 127.0.1.1 10
trace 127.0.1.1
O vizinho é 127.0.1.1
Processando mensagem, que é {'type': 'data', 'source': '127.0.1.1', 'destination': '127.0.1.2', 'payload': '{"type": "trace", "source": "127.0.1.2", "destin
ation": "127.0.1.1", "hops": ["127.0.1.2", "127.0.1.1"]}'}
Mensagem de trace  {'type': 'trace', 'source': '127.0.1.4', 'destination': '127.0.1.2', 'hops': ['127.0.1.4', '127.0.1.3', '127.0.1.2']}
O vizinho é 127.0.1.3
Mensagem de trace  {'type': 'trace', 'source': '127.0.1.4', 'destination': '127.0.1.2', 'hops': ['127.0.1.4', '127.0.1.3', '127.0.1.2']}
O vizinho é 127.0.1.3
```

```
trace 127.0.1.2
O vizinho é 127.0.1.3
Processando mensagem, que é {'type': 'data', 'source': '127.0.1.2', 'destination': '127.0.1.4', 'payload': '{"type": "trace", "source": "127.0.1.4", "destin
ation": "127.0.1.2", "hops": ["127.0.1.4", "127.0.1.3", "127.0.1.2"]}'}
```

Exemplo de funcionamento e comandos feitos no *tmux.*

Após 10 execuções e manipulações diversas da topografia de rede, foi feita uma análise em relação à média de tempo gasto.

	1	2	3	4	5	6	7	8	9	10	$\bar{x}$
fish-e	0,263	0,155	0,154	0,217	0,199	0,363	0,146	0,219	0,170	0,188	2,074
fish-a	1,739	0,171	0,178	0,234	0,138	0,169	0,244	0,213	0,183	1,035	4,304
hub	0,527	0,165	0,184	0,149	0,244	0,217	0,272	0,150	0,136	0,361	2,405



Vemos que a execução da topografia *fish-asymmetric* foi a que de fato teve maior tempo na média e com grandes picos nos testes. A forma *fish-ecmp* me mostrou a mais estável. Entendemos que, embora mudanças leves entre si, todas as topografias estudadas possuem complexidade semelhante e apresentam gráfico relativamente contratante.

5. Conclusão

O projeto foi de suma importância para compreensão do funcionamento e implementação de um sistema de topografia, além de cuidados lógicos como uso de *split horizon*, balanceamento de carga, uso de *tmux* entre vários outros.

Como maior desafio enfrentado, é interessante ressaltar a lógica do comando de deletar – apresentava alta complexidade implementar todas possibilidades de caminhos, recálculo de menor distância, retirada de caminhos antes existentes que passavam por este gateway. Também foi um desafio a análise do código, onde tivemos dificuldade de pensar em parâmetros de valor qualitativo e gerar casos de teste. Foram feitas inúmeras tentativas de criar um sistema de topografia altamente complexo e calcular as diferenças de tempo e processamento de comandos extensivos, porém não apresentaram resultados satisfatórios.

Como aprimoramento, gostaríamos de melhorar o sistema de execução para casos de exceção, que não são completamente tratados.

6. Referências

Split Horizon explained.
https://geek-university.com/ccna/split-horizon-explained/

Distance-vector algorithm.
https://cseweb.ucsd.edu/classes/fa11/cse123-a/123f11_Lec9.pdf

Routing protocol.
https://www.geeksforgeeks.org/computer-network-routing-protocols-set-1-distance-vector-routing/

Quick and easy guide to tmux.
https://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/

A Gentle Introduction to tmux.
https://hackernoon.com/a-gentle-introduction-to-tmux-8d784c404340