TP2: Premium Router

Nome: Artur Henrique Marzano Gonzaga **Matrícula:** 2016006263

Nome: Tiago de Rezende Alves **Matrícula:** 2016006948

1 Introdução

No segundo trabalho prático da disciplina de Redes de Computadores foi solicitado o desenvolvimento de programas roteadores simulando o protocolo de roteamento por vetor de distância.

Em cada roteador deveria ser cadastrado, em tempo de execução, seus enlaces virtuais para outros roteadores. Deveria ser associado a cada enlace virtual o custo de se enviar tráfego através dele. Foi requisitado que os roteadores divulgassem aos roteadores vizinhos as rotas conhecidas por meio de atualizações periódicas e que o roteador implementasse balanceamento de carga (envio uniforme de tráfego quando houverem rotas candidatas com o mesmo custo), *split horizon* (otimização para evitar o problema de *contagem ao infinito*), rerroteamento imediato (envio por rotas candidatas caso um dos enlaces preferenciais seja removido em algum ponto da rota) e remoção periódica de rotas desatualizadas.

Além dos comandos de configuração de enlaces virtuais, os roteadores deveriam suportar um comando *trace* para rastrear a rota para um destino. Foram implementadas também funções *extra*, além das requisitadas na especificação: comandos adicionais na entrada padrão, exibição gráfica da topologia, mensagens de erro para rotas inexistentes e TTL (*time-to-live*) nos pacotes enviados.

1.1 Roteamento por vetor de distância

O protocolo de roteamento solicitado parte do princípio que cada roteador conhece os seus vizinhos – aqueles para os quais ele possui um enlace – e sabe se seus enlaces estão funcionando, assumindo assim que o custo para cada vizinho é conhecido desde o início da operação. Além disso, cada roteador possui uma tabela, denominada *tabela de roteamento*, que mantém as tuplas *(destino, vizinho, custo)* indicando (a) um vizinho direto que alcança o destino e (b) o custo para um pacote chegar ao destino através do vizinho dado. Essa tabela será preenchida incrementalmente, à medida que cada roteador receber mensagens de atualização dos demais.

Cada roteador irá, a cada π segundos (um período pré-configurado), enviar uma mensagem de atualização para os roteadores vizinhos contendo as *menores* rotas que conhece para alcançar todos os nós que conhece da topologia. Os vizinhos, então, comparam as rotas recebidas com as que já conhecem e adicionam em suas tabelas aquelas que possuírem custo menor do que as já conhecidas – ou aquelas que tiverem como destino algum roteador para o qual ainda não se possui rotas. A informação acerca do vizinho que a enviou permite a deleção de rotas, pois ao identificar em uma mensagem de atualização que uma das rotas anteriormente conhecidas passou a não ser divulgada, conclui-se que essa não existe mais, excluindo-a da tabela.

Roteadores	Custos
127.0.0.1	10

Table 1: Tabela de roteamento do 127.0.0.4 ao iniciar.

Roteadores	Custos
127.0.0.1	10
127.0.0.2	20
127.0.0.3	20
127.0.0.5	20

Table 2: Tabela de roteamento do 127.0.0.4 após primeira iteração.

As tabelas 1 e 2 mostram uma iteração do algoritmo de roteamento sobre a topologia em estrela presente na especificação e ilustrada na figura 5. Ao iniciar o roteador, são conhecidos apenas seus vizinhos, motivo pelo qual existe rota apenas para o `127.0.0.1` na tabela 1. Após a primeira iteração o roteador `127.0.0.1`, que conhece a topologia completa, compartilha suas rotas com seus vizinhos, incluindo o `127.0.0.4`, que passará a conhecer toda a topologia.

2 Detalhes de Implementação

Utilizamos o *Python 3.7* em razão da facilidade em criar e gerenciar abstrações, o que favoreceu um maior aproveitamento do tempo durante o desenvolvimento e permitiu que o trabalho fosse desenvolvido de maneira mais fluida. A implementação do trabalho foi dividida em três classes: `Message`, que contém a definição do formato das mensagens a serem trocadas via UDP, `RoutingTable`, que possui as funções associadas ao acesso e manipulação da tabela de roteamento, e *Router*, a classe principal que agrega todas as rotinas referentes ao funcionamento do roteador.

Os roteadores são associados a endereços IP locais na sub-rede `127.0.1.0/24` e utilizam soquetes UDP para se comunicarem entre si. Todos os roteadores usam a porta 55151 para a comunicação. Há também uma interface de linha de comando onde 3 comandos básicos, além de comandos adicionais que implementamos, são suportados:

1. **add** <**ip**> <**weight**>, referente a adição de um novo enlace com peso <`weight`> entre o roteador corrente e aquele associado ao endereço <`ip`>;
2. **del** <**ip**>, que removia o enlace entre o roteador corrente e aquele associado ao endereço <`ip`>;
3. **trace** <**ip**>, que rastreia a rota para o endereço <`ip`>.

A recepção concorrente tanto de mensagens da rede quanto comandos da entrada padrão foi feita com uso de multiplexação de I/O com a biblioteca *selectors* a fim de simplificar o trabalho – caso tivéssemos usado *threads*, teríamos que tratar condições de corrida com primitivas de sincronização, o que seria um esforço desnecessário.

2.1 Atualizações periódicas

As atualizações periódicas de rotas foram implementadas com o uso do *Timer* da biblioteca *threading*, o que permitiu que esse processo fosse feito em *threads* à parte, tornando-o independente do fluxo principal do programa.

2.2 Remoção de rotas desatualizadas

A fim de não comprometer a capacidade de armazenamento do roteador, foi solicitado que rotas fossem removidas da tabela de roteamento caso nenhuma informação sobre elas fosse recebida em um período de 4π, sendo π o período de envio de mensagens de `update`. Em cada roteador, a nossa abordagem utiliza um *timer* para cada vizinho para apagar as rotas desse vizinho. Um *timer* será resetado caso as rotas de um vizinho sejam informadas antes do tempo de remoção. A decisão de usar um *timer* por vizinho se deve ao fato de que, se uma rota estiver desatualizada, então todas as demais rotas informadas pelo vizinho também estarão.

2.3 Exclusão mútua

Utilizamos uma *lock* para resolver problemas de acesso concorrente à tabela de rotas. A *lock* é adquirida sempre que sempre que (a) um comando deve ser processado, (b) uma mensagem da rede deve ser processada, (c) uma mensagem de atualização está prestes a ser enviada e (d) rotas desatualizadas estão prestes a serem removidas. Em cada método que adquire a *lock*, a *lock* é liberada logo antes do final da execução do método.

2.4 Split horizon

Um problema comum no protocolo por vetor de distância é o problema da *contagem ao infinito* (loops de roteamento). Um exemplo clássico desse problema ocorre quando um enlace é removido, mas uma das pontas dele recebe uma atualização de algum outro roteador ainda desatualizado indicando a existência de uma rota através do enlace removido. Enquanto esse problema ocorre, o custo da rota falsa aumenta continuamente na tabela de roteamento da ponta afetada.

O método conhecido como *split horizon* busca diminuir a ocorrência desse problema. O método consiste em fazer com que um roteador A não anuncie ao roteador B as rotas que passam por B ou que tenham B como destino. Dessa forma, são eliminados todos os *loops* de roteamento sobre pares de nós adjacentes, o que diminui a probabilidade de ocorrência desse problema – reduzindo-o a *loops* entre 3 ou mais nós.

2.5 Balanceamento de carga

Outra função implementada diz respeito ao balanceamento de carga entre as rotas de mesmo custo, buscando evitar congestionamento na rede. Após analisar as possibilidades de implementação, escolhemos realizar uma escolha aleatória dentre as rotas de menor custo disponíveis, por ser simples de implementar, computacionalmente eficiente e probabilisticamente eficaz – o tráfego tende a ser igualmente distribuído entre as rotas quando o número de pacotes a serem roteados tender ao infinito.

2.6 Rerroteamento imediato

Um roteador, ao perder uma rota ótima para um destino, deve ser capaz de escolher uma rota alternativa de menor custo dentre as demais conhecidas por ele – caso exista – para encaminhar os pacotes. Para tal, foi necessário que o roteador fosse capaz de armazenar, para cada destino, a menor rota através de cada vizinho que informou uma rota para esse destino, mesmo que nem todas sejam ótimas. Isso foi feito por meio de duas estruturas: (1) um dicionário *vizinho* $\Rightarrow$ *custo* que armazena os enlaces para os vizinhos e (2) um dicionário *destino* $\Rightarrow$ *vizinho* $\Rightarrow$ *custo*, onde cada destino mapeia para um dicionário de vizinhos por onde o pacote pode ser encaminhado, e cada um desses mapeia para o custo total da menor rota. Essa estrutura permitiu a consulta eficiente das melhores rotas.

3 Instruções para execução

O trabalho possui uma única dependência externa, o *graphviz*, para suportar a feature adicional de visualização gráfica da topologia. O diretório do trabalho é munido de um arquivo *Pipfile* (arquivo de configuração do *pipenv*), permitindo a criação automática do ambiente pré-configurado. Com o *pipenv* e o *graphviz* já instalados na distribuição alvo, o ambiente pode ser configurado rodando no diretório raíz:

```
$ pipenv install
```

Depois disso, o ambiente pode ser ativado digitando:

```
$ pipenv shell
```

O programa pode então ser executado normalmente:

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

`<ADDR>` corresponde ao endereço IP do roteador, `<PERIOD>` diz respeito ao intervalo de tempo (π), em segundos, entre as mensagem de atualização do roteador e `<STARTUP>` é um argumento opcional que corresponde ao nome de um arquivo com comandos de configuração a serem executados no início da execução.

Observação. O trabalho também pode ser executado em um ambiente comum, sem nenhuma dependência externa, no *Python 3.7*. Nesse caso, ele executará sem suporte ao comando *plot* – detalhado na seção 5.

4 Testes e análises

Utilizamos os *scripts* fornecidos pelo professor para avaliar a corretude dos aspectos implementados. Com a topologia *hub-and-spoke* foi verificada a corretude das funcionalidades básicas, como a construção das tabelas de roteamento, as mensagens de atualização com *split horizon* e o tratamento das mensagens e dos comandos. A topologia *fish-ecmp* foi utilizada para testar o balanceamento de carga enviando tráfego do nó 1 (cauda) ao 6 (cabeça). Como esperado, constatamos que aproximadamente metade do tráfego segue a rota $1 \Rightarrow 3 \Rightarrow 4 \Rightarrow 6$ e metade segue a rota $1 \Rightarrow 3 \Rightarrow 5 \Rightarrow 6$. A *fish-asymmetric* foi utilizada para verificar a corretude do rerroteamento imediato, removendo o nó 5, por exemplo, e imediatamente enviando tráfego do 1 ao 6. O resultado foi como esperado: ao chegar em 3, o tráfego é encaminhado pela rota alternativa através de 4.

Além dessas, foram criadas novas topologias denominadas *mesh* e *tree* com o intuito de aferir o funcionamento do programá de maneira mais completa.

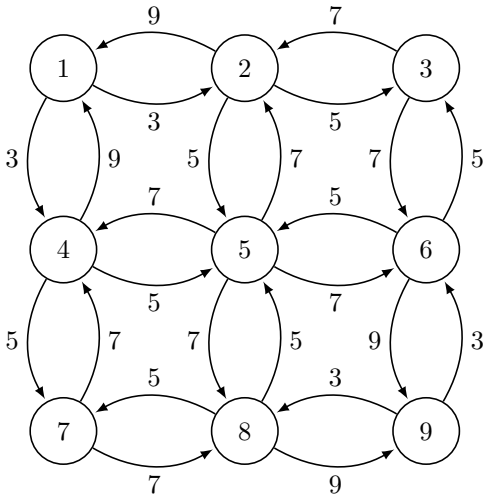

Figure 1: Topologia *mesh*.

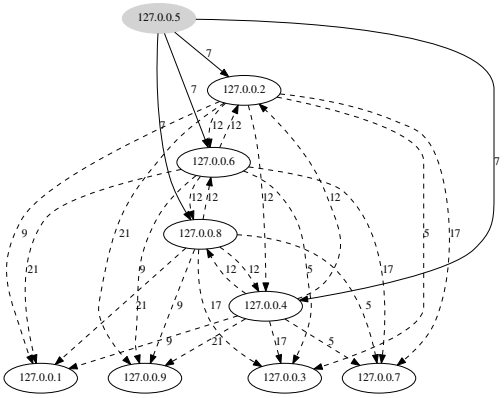

Figure 2: Topologia *mesh* aprendida pelo roteador 5 (127.0.0.5).

Essa topologia possui algumas características interessantes para a execução de testes: além de compacta, possui um alto grau de assimetria e apresenta mais de uma rota ótima para determinados pares de origem e destino. Essas características permitiram o teste de aspectos específicos da implementação, além do funcionamento geral do algoritmo.

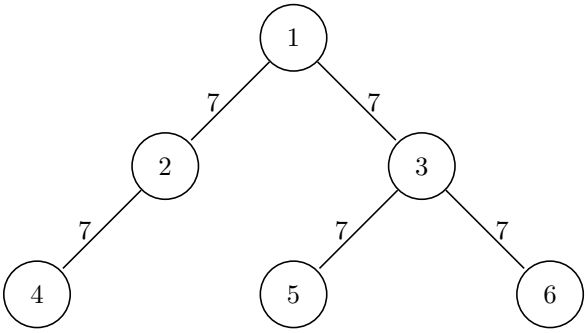

Figure 3: Topologia *tree*.

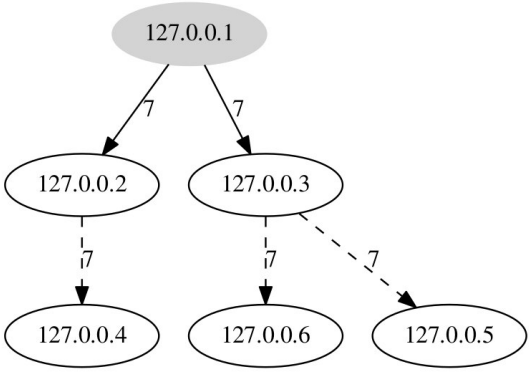

Figure 4: Topologia *tree* aprendida pelo roteador 1 (127.0.0.1).

Os testes realizados na topologia *tree* foram um complemento àqueles realizados na topologia estrela presente na especificação do trabalho e foram em sua maioria relacionados a desconexões e alterações na topologia em geral com o intuito de comprovar o correto funcionamento da lógica aplicada durante a implementação do trabalho. É possível subir uma das topologias do teste com o script *runscript.sh* no diretório raíz, passando o nome do teste (*fish-ecmp*, *fish-asymmetric*, *hub-and-spoke*, *mesh* ou *tree*):

```
$ ./runscript.sh <nometeste> <updatetime>
```

5 Itens adicionais

Com o intuito de aprimorar ainda mais o roteador, algumas funções adicionais, além daquelas exigidas na especificação, tiveram sua implementação realizada.

5.1 Comandos de *debugging*

Para facilitar o *debugging* do trabalho foram implementados os seguintes comandos adicionais:

- `update` – envia uma mensagem de *update* para os vizinhos imediatamente;
- `data <ip> <conteúdo>` – envia um pacote de dados para `<ip>`;
- `routes` – mostra a tabela de roteamento;
- `links` – mostra a tabela de enlaces diretos;
- `time` – mostra o tempo restante para o próximo `update`;
- `clear` – limpa a tela;
- `plot` – gera a representação gráfica da topologia da rede (mais detalhes na próxima seção).

5.2 Visualização de enlaces e rotas

Utilizamos a biblioteca `graphviz` para plotar a topologia a partir de um roteador qualquer, o que permite conferir os enlaces e rotas de maneira mais clara e intuitiva. Para utilizar o recurso basta usar o comando de teclado `plot` e um arquivo com a topologia será salvo em `dot/<ip-roteador>.pdf`, a partir do diretório raíz.

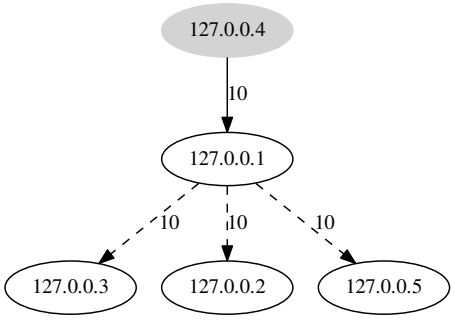

Figure 5: Topologia estrela aprendida pelo roteador 4 (127.0.0.4). Cinza indica o roteador origem; as setas contínuas indicam enlaces e as setas pontilhadas indicam rotas.

5.3 Mensagem de erro para rota inexistente

Conforme sugerido na especificação, também foi implementada uma mensagem a respeito de erros de roteamento. Quando um roteador não encontra rota na sua tabela para encaminhar um pacote, ele envia um pacote de dados informando à origem que não há rota até o destino. Por exemplo, se em alguma topologia o roteador 1 enviar uma mensagem ao 7, o pacote eventualmente passar pelo 5 e ele verificar que não há rota para o 7 em sua tabela, o 5 retorna uma mensagem com o seguinte conteúdo para o 1:

```
{
    "type": "data",
    "source": "127.0.1.5",
    "destination": "127.0.1.1",
    "payload": "No route to 127.0.1.7.",
    "ttl": 128
}
```

5.4 *Time to Live* (TTL)

Outra preocupação foi em relação ao comportamento do sistema durante a estabilização após ocorrerem mudanças na topologia da rede, mais especificamente quando um roteador é completamente desconectado. Um problema desse tipo foi observado na topologia *fish-asymmetric*: suponha que desconectamos o roteador 6 do grafo e enviamos um pacote do roteador 1 ao 6 antes que a informação propague. Como 3 ainda não deletou sua rota para 6 através de 4 ou 5, 3 encaminha o pacote através de um desses enlaces. Como nem 4 nem 5 já deletaram a rota para 6 que passa por 3 (a rota que "dá a volta"), o pacote é retornado para 3, ficando em *loop* entre os roteadores 3, 4 e 5. Com o intuito de evitar que isso aconteça em casos desse tipo, foi implementado o mecanismo de *Time to Live* (TTL), que consiste em um campo incluído em mensagens de *trace* e *dados* com um valor padrão razoavelmente alto (128) que é decrementado toda vez que a mensagem é encaminhada. Quando o TTL se torna igual à zero, o roteador descarta a mensagem e envia uma mensagem à origem indicando que o TTL foi excedido:

```
{
    "type": "data",
    "source": "127.0.0.4",
    "destination": "127.0.0.1",
    "payload": "TTL exceeded.",
    "ttl": 128
}
```

6 Conclusão

Consideramos que o trabalho foi concluído com sucesso. O desenvolvimento do trabalho foi facilitado com os comandos de *debugging* e a visualização de enlaces e rotas, itens adicionais implementados que facilitaram a verificação do correto funcionamento do código. Os itens que geraram um atraso na concepção e nos primeiros passos do desenvolvimento estavam ligados ao uso de *threads* para o tratamento de comandos via terminal e mensagens via UDP e onde associar os *timers* para a deleção de rotas desatualizadas. Essas dúvidas foram resolvidas no decorrer do tempo com a utilização da biblioteca `selectors` para o primeiro problema e com a associação dos *timers* aos roteadores vizinhos no segundo, conforme exposto na documentação. O trabalho foi essencial para fixar conhecimentos acerca do funcionamento do processo de roteamento e de soluções e problemas recorrentes no design de aplicações para a camada de rede. O trabalho também nos situou bem em aspectos de performance, como o balanceamento de carga e o rerroteamento imediato.