DCCRIP: Protocolo de Roteamento por Vetor de Distância

Última modificação: 27 de setembro de 2018

1 Introdução

Neste trabalho iremos desenvolver o DCCRIP, um roteador que utiliza roteamento por vetor de distância. O DCCRIP tem suporte a pesos nos enlaces, balanceamento de carga, medição de rotas e outras funcionalidades.

2 Topologia Virtual

Você deve implementar um programa que emulará um roteador em uma topologia virtual. Cada roteador será associado (`bind`) a um endereço IP local; em particular, iremos criar endereços IP na subrede 127.0.1.0/24 na interface de *loopback* (`lo`) de seu computador. Cada roteador utilizará um soquete UDP associado ao seu respectivo endereço para trocar mensagens de roteamento e dados com outros roteadores (associados a outros endereços IP).

Seu programa deve ter uma interface de linha de comando e suporte a dois comandos de gerência da topologia de enlace virtual:

`add <ip> <weight>` — Este comando adiciona um enlace virtual entre o roteador corrente e o roteador associado ao endereço `<ip>`. Ao calcular rotas com menor distância, o peso de transmitir no enlace virtual (do roteador corrente para o roteador associado ao endereço `<ip>`) é dado por `<weight>`.

`del <ip>` — Este comando remove o enlace virtual entre o roteador corrente e o roteador associado ao endereço `<ip>`.

Enlaces virtuais serão utilizados para transmitir dados. Os pesos serão utilizados no cálculo das rotas mais curtas. Os comandos acima servem para configurar a topologia virtual; além disso, eles podem ser utilizados durante a execução de testes para emular falhas de rede ou mudar a topologia.

2.1 Criando Endereços IP na Interface de *Loopback*

Você pode adicionar um endereço IP a uma interface usando o comando abaixo. Note que o endereço IP deve conter o prefixo identificando a rede à qual a interface está conectada:

```
ip addr add <ip>/<prefixlen> dev <interface>
```

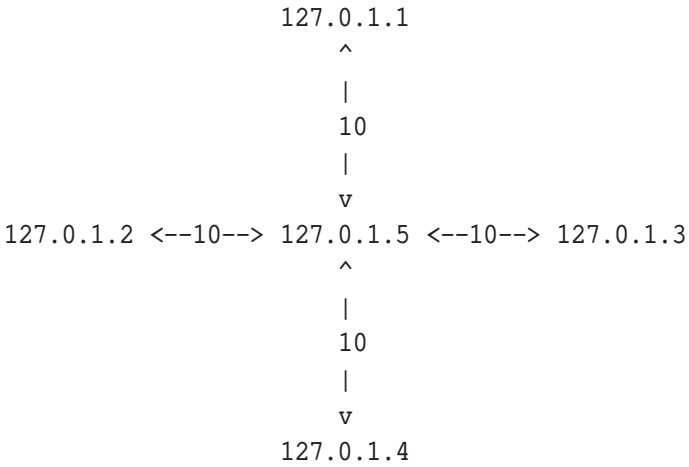
Para adicionar endereços IP à interface de *loopback* no contexto deste trabalho, executaremos comandos como abaixo. O material disponibilizado para o desenvolvimento do trabalho inclui o *script* `lo-addresses.sh`, que executa os comandos necessários para adicionar e remover endereços da interface de *loopback*.

```
ip addr add 127.0.1.1/32 dev lo
ip addr add 127.0.1.2/32 dev lo
...
ip addr add 127.0.1.16/32 dev lo
```

Neste trabalho iremos executar programas (roteadores) que se associam (`bind`) a diferentes endereços IP e trocam mensagens por soquetes de rede. Este modo de operação é equivalente ao modo de operação entre programas executando em computadores distintos.

2.2 Exemplo

Neste trabalho iremos utilizar a topologia virtual abaixo como exemplo:



Para criarmos esta topologia, precisamos executar os seguintes comandos na interface interativa do programa associado ao endereço 127.0.1.5:

```
add 127.0.1.1 10
add 127.0.1.2 10
add 127.0.1.3 10
add 127.0.1.4 10
```

Em todos os outros dispositivos, precisamos executar o comando abaixo:

```
add 127.0.1.5 10
```

3 Codificação de Mensagens de Roteamento

Neste trabalho, todas as mensagens serão codificados utilizando JSON,[1] um formato para codificação de dados amplamente utilizado em aplicações Web. Você pode utilizar bibliotecas para codificação e decodificação de dados para o formato JSON no desenvolvimento deste trabalho. Alunos desenvolvendo o trabalho em linguagens compiladas como C e C++ devem documentar o processo de compilação e ligação com as bibliotecas utilizadas. O formato JSON

[1] `https://www.json.org/`

utiliza apenas texto e pode ser facilmente inspecionado por humanos, o que deve facilitar o desenvolvimento do trabalho.

Todas as mensagens trocadas neste trabalho são dicionários (JSON *objects*) com pelo menos três campos:

`source` — Especifica o endereço IP do programa que originou a mensagem.

`destination` — Especifica o endereço IP do programa destinatário da mensagem.

`type` — Especifica o tipo da mensagem, sua semântica, e quaisquer campos adicionais. Neste trabalho implementaremos três tipos de mensagem: `data` (seção 3.1), `update` (seção 4.1) e `trace` (seção 5).

3.1 Mensagens de Dados

Mensagens de dados devem ter o campo `type` preenchido com o *string* "`data`". Mensagens de dados contêm os três campos acima e um campo `payload`, cujo valor é um *string* qualquer. Seu programa deve imprimir na tela o `payload` de todas as mensagens `data` que receber. Abaixo mostramos um exemplo de mensagem de dados:

```
{
  "type": "data",
  "source": "127.0.1.2",
  "destination: "127.0.1.1",
  "payload": "{\"destination\": \"127.0.1.2\", \"type\": \"trace\", ...}"
}
```

4 Roteamento por Vetor de Distância

Seu roteador deve implementar um protocolo de roteamento por vetor de distância. Seu roteador deve enviar uma mensagem de atualização para seus vizinhos (seção 4.1), manter informações de rotas em uma tabela de roteamento e calcular rotas mais curtas para encaminhamento de dados (seção 4.2).

4.1 Mensagens de Atualização de Rotas

Seu roteador deve enviar uma mensagem de atualização de rotas, chamada também de `update`, periodicamente para cada um de seus vizinhos. Um vizinho é um outro roteador que está diretamente conectado. Vizinhos são adicionados pelo comando `add` e removidos pelo comando `del` da interface iterativa de teclado (seção 2).

Mensagens de `update` devem ter o campo `type` preenchido com *string* "`update`". Os campos `source` e `destination` devem ser preenchidos com o endereço IP do roteador corrente e do roteador vizinho destinatário, respectivamente.

Mensagens de `update` devem ter um quarto campo `distances` contendo um dicionário informando ao vizinho as melhores rotas conhecidas pelo roteador corrente. Mais especificamente, o campo `distances` deve ser um dicionário contendo pares chave-valor mapeando um endereço IP de destino à menor distância conhecida para alcançar aquele destino.

Atualizações Periódicas. Seu roteador deve enviar mensagens de `update` periodicamente para todos seus vizinhos a cada π segundos. O valor de π é constante durante uma execução do roteador, mas pode ser alterado entre diferentes execuções através de um parâmetro de linha de comando.

Split Horizon. As mensagens de `update` devem implementar a otimização *split horizon*. Mensagens de `update` enviadas ao vizinho *x não* devem incluir rotas para *x* e rotas aprendidas de *x*. Esta otimização reduz a probabilidade de ocorrência do problema de contagem ao infinito.

4.1.1 Exemplo

Abaixo mostramos uma mensagem de `update` enviada do roteador 127.0.1.5 na topologia exemplo mostrada na seção 2.2.

```
{
  "type": "update",
  "source": "127.0.1.5",
  "destination": "127.0.1.1",
  "distances": {
    "127.0.1.4": 10,
    "127.0.1.5": 0,
    "127.0.1.2": 10,
    "127.0.1.3": 10
  }
}
```

4.2 Tabela de Roteamento e Encaminhamento de Dados

Seu roteador deve armazenar informações sobre todas as rotas conhecidas em uma tabela de roteamento. Seu roteador deve utilizar as informações disponíveis na tabela de roteamento para encaminhar dados a um destino através da rota mais curta. O encaminhamento de dados deve implementar as seguintes funcionalidades:

Balanceamento de Carga. Se múltiplas rotas estiverem empatadas com a menor distância até um destino, seu roteador deve dividir o tráfego uniformemente entre as rotas. Na topologia de exemplo, se criarmos um enlace de peso 20 entre os roteadores 127.0.1.1 e 127.0.1.2, teríamos duas rotas empatadas com distância 20. Neste caso, metade do tráfego entre estes roteadores deve passar pelo enlace direto e metade deve passar pelo enlace através do roteador 127.0.1.5.

Reroteamento Imediato. Quando a rota mais curta para um destino deixa de existir (por exemplo, por que o enlace para o vizinho foi removido utilizando o comando `del`), seu programa deve encaminhar dados através da rota remanescente de menor custo. Em outras palavras, a remoção de um enlace da topologia não deve impactar encaminhamento de dados caso existam rotas alternativas.

Remoção de Rotas Desatualizadas. Seu programa deve remover da tabela de roteamento informações de rotas que não foram informadas por um período de tempo igual a 4π, onde π é o período entre envio de mensagens de `update` (seção 4.1).

O roteador deve descartar mensagens destinadas a endereços pros quais o roteador não conhece nenhuma rota. Valendo pontos extras, alunos podem implementar funcionalidade na qual um roteador envia uma mensagem de erro informando à origem que não existe rota até o destino.

5 Mensagens de rastreamento

Além das mensagens de `data` e `update`, seu roteador deve processar mensagens de `trace`. Mensagens de `trace` são utilizadas para medir a rota utilizada entre dois roteadores na rede. Além dos campos `type`, `source` e `destination`, mensagens de `trace` possuem um campo `hops`, que armazena a lista de roteadores por onde a mensagem de trace já passou.

Ao receber uma mensagem de `trace`, o roteador deve adicionar seu endereço IP ao final da lista no campo `hops` da mensagem. Após isso, o roteador deve verificar se é o destino do `trace`. Se o roteador *não* for o destino do `trace`, ele deve encaminhar a mensagem através de um dos caminhos mais curtos que conhece para o destino. (Mensagens de `trace` estão sujeitas a balanceamento de carga.)

Se o roteador for o destino do `trace`, ele deve enviar uma mensagem `data` para o roteador que originou o `trace`; o `payload` da mensagem `data` deve ser um *string* contendo o JSON correspondente à mensagem de `trace`. Em outras palavras, o resultado do `trace` deve ser enviado no `payload` de uma mensagem `data` até o solicitante. Um exemplo de mensagem de dados contendo o resultado de um `trace` é mostrado na seção 3.1.

Um roteador deve criar uma mensagem de rastreamento para o roteador com endereço `<ip>` quando receber um comando de teclado com o formato abaixo:

```
trace <ip>
```

5.1 Exemplo

Abaixo mostramos uma mensagem de `trace` originada pelo roteador 127.0.1.1 e recebida pelo roteador 127.0.1.2 na topologia de exemplo. O roteador 127.0.1.2 deve encapsular o resultado do `trace` em uma mensagem de dados, como mostrado na seção 3.1.

```
# Mensagem recebida por 127.0.1.2:
{
  "type": "trace",
  "source": "127.0.1.1",
  "destination": "127.0.1.2",
  "hops": ["127.0.1.1", "127.0.1.5"]
}

# Resultado do trace enviado como payload a 127.0.1.1:
{
  "type": "trace",
  "source": "127.0.1.1",
  "destination": "127.0.1.2",
  "hops": ["127.0.1.1", "127.0.1.5", "127.0.1.2"]
}
```

6 Implementação e Interface com o Usuário

Você deverá implementar o DCCRIP usando soquetes UDP.[2] Seu roteador deve interoperar com outros emuladores; teste com roteadores de colegas e com o exemplo disponibilizado no pacote de testes. O trabalho pode ser implementado em Python, C, C++, Java, ou Rust, mas deve interoperar com emuladores escritos em outras linguagens.

6.1 Inicialização

O roteador DCCRIP deve ser inicializado como segue:

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

Onde `ADDR` indica a qual endereço IP o roteador deve se associar (`bind`) e `PERIOD` representa o período entre envio de mensagens de `update` (π). O parâmetro `STARTUP` é opcional e deve ser o nome de um arquivo contendo comandos de teclado para adicionar links. Arquivos de `STARTUP` são utilizados para montar a topologia virtual no início da execução dos roteadores. Alternativamente, seu roteador pode receber os parâmetros `ADDR`, `PERIOD` e `STARTUP` através de opções de linha de comando chamadas `--addr`, `--update-period` e `--startup-commands`, respectivamente (como na implementação do professor). Todos os roteadores devem usar a porta 55151 para comunicação.

6.2 Execução e Terminação

Seu roteador deve ler comandos `add`, `del` e `trace` (seção 2) da entrada padrão, isto é, digitados do teclado. Cada roteador só deve imprimir na saída padrão o conteúdo (`payload`) de mensagens de dados endereçadas a ele. Se você precisar imprimir mensagens para auxiliar a depuração, utilize algum arquivo ou biblioteca de *logging*. Todos os roteadores devem utilizar a porta 55151.

O roteador deve parar de executar quando receber um sinal de interrupção de teclado (`ctrl-c`) ou quando receber o comando `quit` do teclado.

7 Avaliação

Qualquer incoerência ou ambiguidade na especificação deve ser apontada para o professor; se confirmada a incoerência ou ambiguidade, o aluno que a apontou receberá um ponto extra.

7.1 Grupos

Este trabalho pode ser executado em grupos de até dois alunos (todos os alunos são responsáveis por dominar todos os aspectos do trabalho).

7.2 Entrega

O programa deve ser entregue como apenas um arquivo fonte na linguagem escolhida (`router.c`, `router.cpp`, `router.py`, `router.java`, `router.rs`). Para programas em linguagens compiladas,

[2] Utilize `socket(AF_INET, SOCK_DGRAM, 0)`, ou equivalente, para criar o soquete.

a entrega deve conter documentação ou metadados (por exemplo, um `Makefile`) explicando como compilar o programa. Além disso, deve ser entregue também um arquivo PDF de documentação, descrito a seguir.

7.3 Documentação

Cada grupo deverá entregar documentação em PDF de *até* 6 páginas (três folhas), sem capa, utilizando fonte tamanho 10, e figuras de tamanho adequado ao tamanho da fonte. A documentação deve discutir desafios, dificuldades e imprevistos de projeto, bem como as soluções adotadas para os problemas. Em particular, sua documentação deve discutir, utilizando trechos de código, como foram implementadas as seguintes funcionalidades:

- Atualizações Periódicas
- *Split Horizon*
- Balanceamento de Carga
- Reroteamento Imediato
- Remoção de Rotas Desatualizadas

7.4 Testes

Uma implementação de referência e um conjunto de testes será disponibilizado no Moodle. Testes complementares serão executados durante a correção do trabalho.