Documentação TP2

Gabriel Soares
Mariane Fernandes

10 de Dezembro de 2018

1 Introdução

O objetivo deste trabalho é implementar o protocolo RIP, que é baseado em vetor de distâncias. O algoritmo de vetor de distâncias funciona baseado na premissa de quê, tudo que um nó sabe sobre a rede, ele repassa para seus vizinhos imediatos.

Para esta implementação, foi simulada uma topologia de roteadores na interface de loopback, onde cada um dos roteadores na topologia é representado por um endereço na subrede 127.0.1.0/24. Para minimizar problemas de contagem para o infinito, o protocolo deve implementar a otimização de *split horizon*, que impede que um nó responda a um vizinho as mesma informações que acabou de receber deste.

2 Mensagens

As mensagens trocadas entre os roteadores descrevem arquivos JSON, e possuem três tipos: mensagens de dados, comando trace e mensagens de update. Todos três tipos obrigatóriamente possuem os campos *tipo, source, destination*. Abaixo estão descritos os campos específicos de cada tipo, além dos campos obrigatórios:

- Mensagens de Dados: possuem o campo *payload*, que é uma string qualquer que contém os dados úteis que estão sendo enviados.
- Comando Trace: possuem o campo *hops*, que descreve os passos percorridos pelo pacote - endereços de IP dos roteadores no caminho - em uma determinada rota.
- Mensagens de Update: possuem o campo *distances*, onde são descritos os destinos cuja distância está sendo atualizados, seguidos dos seus novos pesos. Mensagens de uptdate devem ser enviadas de acordo com uma periodicidade constante, definida no início da execução do programa.

3 Código

O código está contido num único arquivo, *router.py*. A execução do código é feita de acordo com o solicitado na especificação

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

Onde ADDR é o endereço de IP ao qual o roteador em questão deve fazer o *bind*, PERIOD é o tempo estipulado para as atualizações periódicas e STARTUP é um parâmetro opcional, que faz referência a um arquivo com configurações iniciais para o roteador. Para cada roteador na topologia, uma instância deve estar executando *router.py* Após o início da execução do código, a interface de linha de comando suporta as seguintes entradas:

- *add IP PESO* que adiciona um roteador de IP à topologia com custo PESO
- *del IP* de deleta um roteador IP da topologia
- *quit* que desliga o roteador que está sendo executado na instância

3.1 Vetor de Distâncias

O principal objetivo deste trabalho é fazer uma implementação do protocolo de roteamento que utiliza vetor de distâncias. O princípio básico de funcionamento é que cada nó da rede repasse aos seus vizinhos imediatos a informações que ele conhece sobre a rede. No código fonte, as principais implementações para o vetor de distâncias estão no método adicionarDados() que monta a tabela do vetor de distâncias e o método rotearVetor(), que monta um pacote e repassa as informações da tabela aos vizinhos.

O método adicionarDados() primeiro verifica se o destino já está listado na tabela, e o inclui caso não esteja. Caso o destino já esteja listado, verifica-se se a entrada em questão é uma rota alternativa, e em caso positivo, o peso da nova rota é verificada. Se a rota alternativa é uma rota pior que a existente, ela é desconsiderada; se a rota alternativa é uma rota melhor, ele substitui a rota existente; e se a rota alternativa é uma rota de mesmo custo, ele é adicionada ao vetor de distâncias e será utilizada para balancear a cargar de trafégo. A função também verifica se a rota recebida já está registrada e, caso já esteja, atualiza o *timestamp* da rota.

3.2 Rotas Desatualizadas

Ao adicionar uma nova rota, a função adicionarDados() chama a iniciaTemporizador() para fazer o monitoramento da nova rota adicionada. Esse monitoramento é feito por uma *thread* que verifica o *timestamp* da rota de acordo com o período definido na inicialização. Caso a rota não tenha sido vista em um pacote do tipo *update* dentro do perído, a rota é retirada. As rotas adicionadas pelo usuário recebem um tratamento diferente pois não são atualizadas pelos vizinhos. Sendo assim, caso a rota utilizada seja uma rota adicionada pelo usuário, ela terá o *timestamp* atualizado para evitar sua remoção por falta de atualização.

```
def iniciaTemporizador(self, destino, caminho):
  threadTemporizador = threading.Thread(target = self.supervisionarTempo,
    args = [destino, caminho])
  threadTemporizador.start()

def supervisionarTempo(self, destino, caminho):
```

```
    existeRota = True
    tempoPercorrido = 0

    while existeRota and self.ligado:
      existeRota = False
      rotas = None

      with self.permissaoMapa:
        if destino in self.mapa:
          rotas = self.mapa[destino].copy()

      if rotas:
        for rota in rotas:
          atualizou = ((time.time() - rota.timeStamp) < 4*self.period and
            rota.destino == destino and
            rota.caminho == caminho)
          fixo = (destino in self.linkFixo and
            self.linkFixo[destino].peso == rota.peso and
            self.linkFixo[destino].caminho == caminho)

          if fixo or atualizou:
            if fixo:
              rota.timeStamp = time.time()

            existeRota = True
            tempoPercorrido = time.time() - rota.timeStamp

      if existeRota:
        time.sleep(4*self.period - tempoPercorrido)

    self.removerDados(destino, caminho)
```

3.3 Atualizações periódicas e Split Horizon

No Algoritmo, cada nó deve enviar atualizações periódicas aos seus vizinhos para sinalizar que ele ainda está ativo. No método rotearVetor(), sempre que o roteador está ligado, aguarda o tempo determinado na linha de comando, e após esse período, envia uma mensagem de update.

Dentro deste método também é feita a restrição do Split Horizon, que impede que um nó devolva ao seu vizinho as informações que acabou de receber dele. Esta restrição é essencial para minimizar a possibilidade de contagem para o infito.

```
  def rotearVetor(self):
    while self.ligado:
      time.sleep(self.period)
      dados = None
```

```
with self.permissaoMapa:
  dados = self.mapa.copy()

for endereco in dados:
  if self.ehVizinho(dados[endereco]):
    pacote = {"type": "update", "source": self.HOST, "destination": endereco}
    distances = {}

    for auxEndereco in dados:
      #verifica se o caminho existe e realiza o split horizon
      if (self.existeCaminho(auxEndereco, dados) and
        not self.passaPeloVizinho(endereco, dados[auxEndereco])):
        distances[auxEndereco] = dados[auxEndereco][0].peso

    distances[self.HOST] = 0
    pacote["distances"] = distances
    self.encaminharPacote(pacote)
```

3.4 Balanceamento de carga

No método encaminharPacote() é realizado o processo de balanceamento de carga. Como ele é o único método que envia pacotes, o balanceamento de carga é garantido para todos os envios. Como o roteador guarda um vetor de rotas para cada destino - todas com o mesmo peso - o balanceamento de carga é feito da seguinte forma: todo envio é feito pelo caminho da primeira posição do vetor e após o envio a rota utilizada e colocada na última posição do vetor.

```
def encaminharPacote(self, pacote):

    #altera a ordem da lista para fazer o balanceamento de carga
    if (pacote["destination"] in self.mapa and
      len(self.mapa[pacote["destination"]]) > 1):
      self.mapa[pacote["destination"]] = self.mapa[pacote["destination"]][1:]
       + [self.mapa[pacote["destination"]][0]]
```

3.5 Reroteamento imediato

O reroteamento imediato é usado para permitir que o pacote seja entregue mesmo que algum link do caminho caia durante o envio do pacote. Esta funcionalidade pode ser disparada pelo método removerLink(), que simula a queda de um link removendo seus dados da tabela.

Para garantir essa propriedade, o roteador guarda uma cópia dos links adicionados pelo usuário em um local separado e sem temporizador. Ao detectar que todas as rotas para um destino cairam, o roteador verifica se existe uma rota, adicionada pelo usuário, para o destino e a adiciona a tabela de roteamento. Caso a queda tenha sido causada pelo comando *del*, o roteador irá aprender uma nova

rota pelo algoritmo de vetor de distâncias. Como todos os vetores guardando os dados originais, se existe algum link até o destino ele será adicionado e propagado novamente por algum roteador.

```
def removerLink(self, ip):
  with self.permissaoMapa:
    if ip in self.mapa:
      self.mapa.pop(ip)

  if ip in self.linkFixo and self.ligado:
    self.adicionarDados(self.linkFixo[ip])
```

4 Conclusão

Neste trabalho foi possível ver na prática algumas dificuldades da implementação de algoritmos de roteamento, tal como a resiliência da rede em caso de falha em algum link, evitar loops de comunicação e atualizar a topologia periodicamente. Além dessas particularidades conceituais, um outro desafio, dessa vez técnico, foi a forma de simular a representação de uma topologia de rede utilizando a interface de *loopback* de uma máquina, sendo este um novo conceito a ser praticado. O monitoramento de múltiplas instâncias de execução também representa uma tarefa não-trivial a ser executada.