DCCRIP - Documentação

Marcelo Nunes
Wanderson Sena

1. Introdução

O DCCRIP é um roteador que utiliza o algoritmo de vetor de distâncias para salvar a sua tabela de roteamento. O DCCRIP tem suporte para balanceamento de carga, peso nos enlaces, medição de rotas, split horizon dentre outros. Todas essas funcionalidades serão detalhadas nas seções seguintes.

2. Inicialização

Para a inicialização do programa, são esperados pelo menos dois parâmetros obrigatórios. São eles, o endereço do roteador e a frequência de atualização. Um terceiro parâmetro pode ser informado para definir o script que será utilizado para a criação da topologia inicial. Caso ele não seja informado é utilizada a entrada padrão para esperar a criação manual. Durante o processo de inicialização, o programa define o seu endereço, frequência e entrada de dados de acordo com o que foi informado via linha de comando. Após isso ele cria o seu dicionário de vizinhos e dicionário de roteamento, caso um script tenha sido informado esses dicionários já são populados.

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

Imagem 1: Linha de comando para inicialização do DCCRIP

3. Execução

Durante a execução do DCCRIP, quatro aspectos devem ser levados em consideração. O programa deve, paralelamente, enviar mensagens de atualização, receber mensagens de outros roteadores , verificar se o usuário enviou novos comandos e verificar o tempo de expiração (timeout) das informações que cada vizinho já o enviou. Para isso, a melhor solução foi distribuir cada uma dessas atividades em threads distintas que são executadas em paralelo. A maior dificuldade dessa implementação é garantir o controle de todas as threads. Como elas são independentes, ao executar o comando de terminação, somente a thread que recebeu o comando era encerrada. Para resolver isso, é preciso combinar as três threads em uma novamente, e após isso, encerrar o programa.

Para garantir a execução em paralelo, às atividades principais foram divididas da seguinte forma. A thread principal é responsável por enviar os updates da sua tabela de roteamento e verificar se as rotas conhecidas ainda não foram expiradas. A segunda thread é responsável por receber as mensagens de outros roteadores, podendo ser elas tanto mensagens de dados quanto de controle. A terceira thread possui a função de aguardar mensagens enviadas pelo usuário via linha de comando. Por fim, a quarta thread é executada de tempos em tempos, de acordo com o <period> especificado na execução do programa, sendo responsável por verificar quais os tempos já foram excedidos para remover informações antigas da tabela de roteamento.

Para facilitar compreensão do programa, cada thread será detalhada nos tópicos seguintes.

3.1. Estruturas de dados

Antes de entrar em detalhes a respeito das threads, é importante explicar as duas estruturas de dados criadas durante a execução para montar a topologia virtual de cada roteador. A primeira é o dicionário de vizinhos, ele é responsável por relacionar o ip com o peso dos seus vizinhos diretos.

Esse dicionário possibilita o envio de mensagens diretas, principalmente as de update, para todos os vizinhos diretos do nó.

```
'ip' : { ' weight' , 'indexOfNext' , [ 'next ' ] , [ 'timeout' ] }
'127.0.0.15': { 10, 1, [ '127.0.0.10', '127.0.0.13' ], [ 4, 8 ] }
```

Imagem 2: Estrutura do dicionário de roteamento e exemplo de preenchimento

A segunda é o dicionário de roteamento, essa estrutura tem a função de armazenar os campos *weight* , *indexOfNext* , *next* e *timeout* para representar a tabela de roteamento. O campo *weight* é referente ao peso da rota, *indexOfNext* é um índice auxiliar para o balanceamento de carga, *next* e *timeout* são listas para representar o roteador que deve receber a mensagem e o tempo que aquela rota está ativa. No caso de múltiplas rotas com o mesmo peso, os roteadores são adicionados na lista para que seja possível executar o balanceamento de carga.

4. Thread principal

Como dito anteriormente, essa thread é responsável por enviar mensagens de atualização para os seus vizinhos.

4.1. Envio de atualizações

Para enviar uma atualização da tabela de roteamento é preciso fazer uma cópia da tabela atual, aplicar o algoritmo de *Split Horizon* e enviar essa cópia para o vizinho desejado. A cópia da tabela é necessária pois o *Split horizon* precisa remover caminhos que levam ou que tenham sido informados pelo vizinho ao qual a mensagem se destina. Isso é feito para minimizar o problema de contagem ao infinito. A implementação do algoritmo foi feita conforme a imagem a seguir.

```
#Applying Split Horizon
for old in distances.copy():
    if key == old:
        del distances[old]
    else:
        if key in distances[old]['next']:
            if len(self.routingTable[old]['next']) == 1:
                #if it is unique route in the table, remove it
               del distances[old]
```

Imagem 3: Implementação do *Split Horizon*

É importante ressaltar que esse código está contido dentro de um loop. Sabendo disso, a variável ‘key’ representa um dos vizinhos do roteador e ‘distances’ representa uma cópia da tabela de roteamento. Assim, para cada vizinho do DCCRIP é feita uma cópia da tabela de roteamento e o algoritmo é aplicado em cima da cópia. Caso o vizinho que irá receber a mensagem esteja presente na tabela de roteamento, tanto por ter enviado a rota ou por ser o próximo passo, a entrada é removida. Ao final, a mensagem que é enviada contém todas as entradas que devem se manter atualizadas para aquele vizinho.

5. Segunda thread

Essa thread é responsável por receber as mensagens de outros roteadores. Para isso, O primeiro passo após receber uma mensagem é verificar se ela é do tipo de *update*, *table, traceroute*

ou *data*. Após isso, a mensagem é tratada conforme o seu tipo. Por fim, o roteador espera outra mensagem e o ciclo recomeça.

5.1. Mensagens de *update*

Quando uma mensagem de update é recebida, é preciso verificar se ela foi enviada por um vizinho, se existem novas rotas e se há mudanças nas rotas conhecidas. O primeiro caso garante que somente mensagens de vizinhos imediatos irão alterar a tabela atual evitando erros caso acidentalmente o roteador receba o update de um nó que não é seu vizinho.

```
# If old weight greater than new weight -> swap
# If ip not in routingTable, add
if ((address not in self.routingTable) or
     int(self.routingTable[address]['weight']) >
(int(distances[address]) +
     int(self.neighborsTable[data['source']])) ):

  self.routingTable[address] = {}
  self.routingTable[address]['weight'] = int(distances[address]) +
    int(self.neighborsTable[data['source']])
  self.routingTable[address]['next'] = [data['source']]
  self.routingTable[address]['indexOfNext'] = 0
  self.routingTable[address]['timeout'] = [4*self.period]
```

Imagem 5: Adição de novas rotas e rotas mais curtas

Tendo a certeza que a mensagem é de um vizinho do roteador, e caso haja uma rota que não está presente na tabela atual, ela é adicionada no dicionário de rotas. Devido a implementação da tabela de rotas como um dicionário, esse trecho do código também realiza a atualização da rota caso exista um caminho mais curto. Se existir um caminho alternativo, com o mesmo peso, a partir de outro roteador, uma nova entrada é adicionada na lista de *next* e de *timeout* contendo o próximo passo para aquele caminho.

Após adicionar e atualizar as rotas encontradas, o algoritmo verifica a sua tabela e busca rotas que não foram anunciadas na mensagem. Se ele encontrar rotas não anunciadas, significa que seu vizinho não possui mais esse caminho e a rota em questão deve ser excluída. No caso de haver mais de um caminho para o mesmo destino, apenas o caminho não anunciado é excluído. Isso permite o reroteamento imediato, já que as mensagens continuam sendo enviadas desde que haja um caminho disponível.

```
for old in self.routingTable.copy():
  if data['source'] in self.routingTable[old]['next']:
    if old not in distances:
      if len(self.routingTable[old]['next']) == 1:
        #if it is unique route in the table, remove it
        del self.routingTable[old]
      else:
        #if have more then one, remove just that index from next and timeout
        self.routingTable[old]['next'].remove(data['source'])
        del self.routingTable[old]['timeout'][self.routingTable[old]['next']
            .index(data['source']) ]
```

```
        self.routingTable[old]['indexOfNext'] =
(self.routingTable[old]['indexOfNext']
        + 1) % len(self.routingTable[old]['next'])
    else:
      self.routingTable[old]['timeout'][ self.routingTable[old]['next']
      .index(data['source']) ] = 4*self.period
```

Imagem 6: Remoção de rotas não anunciadas

Por fim, caso a mensagem informe que uma rota em questão foi excluída o roteador remove ela da tabela.

5.2. Mensagens de *table* ou *trace*

Devido às similaridades das mensagens de *table* e *trace*, elas são tratadas pela mesma seção do código. Em ambos os casos é preciso verificar se o roteador é o destino final da mensagem. Caso não seja, o roteador busca o endereço de destino na tabela de roteamento e encaminha a mensagem. Caso o tipo da mensagem seja *trace* o endereço do roteador é adicionado na tabela de *hops*. Caso o roteador não consiga encaminhar a mensagem, uma mensagem de erro é enviada para o roteador de origem.

```
# Payload is the table of this node, if type is table
if(data['type'] == 'table'):
  message.update({ 'payload' : [ (ip,self.routingTable[ip]['next'],self.
  routingTable[ip]['weight']) for ip in self.routingTable ] })

# Payload is the trace generated, if type is trace
if(data['type'] == 'trace'):
  # If i don't know this neighbor yet, i can not ask trace for him.
  if message['destination'] not in self.routingTable:
    message['payload'] = "Unable to communicate with this node."
  else:
    message.update({'payload' : data['hops']})
    message['payload'].append(self.myAddress)
```

Imagem 7: Payload para mensagens de *trace* e *table*

Caso o roteador seja o destino das mensagens, ele envia uma mensagem de dados para a origem. No caso de mensagens *table* o payload contém uma cópia da tabela de roteamento. Para a mensagem de *trace* o payload contém todos os *hops* junto com o endereço do roteador.

5.3. Mensagens de *data*

No caso de mensagens de *data*, o roteador precisa verificar se aquela mensagem é para ele ou se ele deve encaminhar a mensagem para o próximo roteador. Caso ele seja o roteador de destino, o payload da mensagem é exibido na saída padrão. Caso contrário a mensagem é encaminhada ao roteador mais próximo do destino. Ao fazer isso o campo de *indexOfNext* é incrementado para permitir o balanceamento de carga.

6. Terceira thread

A função dessa thread é receber os comandos emitidos pelo usuário. Caso o usuário tenha informado um script de entrada durante a inicialização, o script é executado por inteiro e, após isso, o

programa passa a receber comandos via entrada de dados padrão. Se não houver script de entrada, o programa irá aguardar os comandos via entrada padrão. Essa thread também é responsável por realizar as primeira adições de rotas após a inicialização do programa. Foi implementado também um comando auxiliar '*print*' no qual o roteador imprime na saída padrão a sua tabela de roteamento. Esse comando permite que o usuário tenha uma rápida visualização das rotas conhecidas pelo roteador.

6.1. Comandos *add* e *del*

```
# Add Neighbor
if readCommand.split(' ')[0] == 'add':
  if len(readCommand.split(' ')) <= 2:
    print("Input failure. Correct pattern:\t add <ip> <weight>")
  else:
    if readCommand.split(' ')[1] == self.myAddress:
      continue
    self.routingTable[readCommand.split(' ')[1] ] = {}
    self.routingTable[readCommand.split(' ')[1] ]['weight'] =
readCommand.split(' ')[2]
    self.routingTable[readCommand.split(' ')[1] ]['next'] = [readCommand.split('
')[1]]
    self.routingTable[readCommand.split(' ')[1] ]['timeout'] = [4*self.period]
    self.routingTable[readCommand.split(' ')[1] ]['indexOfNext'] = 0
    self.neighborsTable[readCommand.split(' ')[1] ] = readCommand.split(' ')[2]
    self.sendUpdates(repeat=False)
```

Imagem 8: Função para adicionar vizinhos

Os comandos *add* e *del* são utilizados para popular a o dicionário de roteamento e montar a topologia virtual do roteador. O comando *add* adiciona um vizinho direto na tabela de vizinhos do roteador. Ao adicionar um novo vizinho, é inserida na tabela de roteamento o caminho direto para ele. De forma complementar, a função *del* remove um vizinho direto da tabela de vizinhos do roteador. Após removê-lo, o programa também remove o caminho da tabela de roteamento. Caso haja mais de um caminho, somente o caminho direto é removido. Como a execução desses comandos implica em alterações no dicionário de roteamento, uma mensagem de atualização é enviada após a execução do comando. Essa implementação do roteamento direto permite que os vizinhos sejam informados quando a adições ou remoções no roteador rapidamente.

6.2. Comandos *table* e *trace*

Para os comandos de *table* e *trace*, há uma verificação para garantir que o roteador conhece um caminho para o endereço de destino. Após isso, é montada a mensagem que será enviada. Caso ela seja de *table*, a mensagem é enviada diretamente para o próximo roteador. Caso ela seja de *trace*, o endereço atual do roteador é adicionado há uma tabela de *hops* que é anexada a mensagem. Após isso, a mensagem é enviada para o próximo roteador. Em ambos os casos há a atualização do *indexOfNext* para o balanceamento de carga. Além disso, em ambos a mensagem de resposta é recebida pela segunda thread como já foi detalhado.

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP socket
message = {'type'  : 'table', 'source':self.myAddress , 'destination':
```

```
destination}
#send the request to the next on the routing table
sock.sendto(str.encode ( json.dumps( message ) ) , (
self.routingTable[destination]['next'][
self.routingTable[destination]['indexOfNext'] ], self.port) )
```

Imagem 9: Exemplo de uma mensagem de requisição da tabela de roteamento

7. Quarta thread

A função dessa thread é verificar, se uma rota desatualizada deve ser removida da tabela, e caso não seja, seu tempo de vida é decrementado.

7.1. Verificação de rotas

Para o acompanhamento do tempo de vida das threads, é usada a lista 'timeout' da tabela de roteamento. Com base nela, a função checkAndUpdatePeriods, cuja implementação pode ser vista abaixo realiza as verificações e deleções. Esse algoritmo é executado a cada π segundos, sendo π a frequência informada durante a inicialização.

```
def checkAndUpdatePeriods(self):
  for ip in self.routingTable.copy():
    if ip == self.myAddress : continue # Don't remove my ip
      for time in self.routingTable[ip]['timeout']: # How we have a list of
possible next's (balance load), we have a time for each
        if int(time) <= 0: # if a time of next is over, remove it
          if len(self.routingTable[ip]['next']) == 1:
            del self.routingTable[ip]
            else:
              index = self.routingTable[ip]['timeout'].index(time)
              del self.routingTable[ip]['timeout'][index]
              del self.routingTable[ip]['next'][index]
              self.routingTable[ip]['indexOfNext'] = ( self.routingTable[ip]
              ['indexOfNext'] + 1 ) % len(self.routingTable[ip]['next'])
        else:
          # Update timeout of next
          self.routingTable[ip]['timeout'][ self.routingTable[ip]['timeout']
          .index(time) ] -= self.period

  # Check and decrease each 'period' seconds
  checkAgain = threading.Timer( self.period, self.checkAndUpdatePeriods)
  checkAgain.start()
```

Imagem 4: Algoritmo para remoção de rotas desatualizados

Dessa forma, o algoritmo percorre a sua tabela de roteamento e realiza a seguinte verificação. Se o timeout for menor ou igual a zero, aquela rota está expirada e então ela deve ser removida. Caso contrário, ela ainda é válida, porém é timeout dela é decrementado. Por fim é utilizado um timer para executar a função novamente após π segundos.