Documentação TP2

Marcel Henrique S. Mendes

[1]Departamento de Ciência da Computação (DCC) - Universidade Federal de Minas Gerais

marcelmendes@dcc.ufmg.br

1. Descrição geral

Neste segundo trabalho prático foi pedido pelo professor Ítalo Cunha a implementação de um Roteador (simulador) que opera sob o algoritmo vetor de distâncias. Cada roteador mantêm uma tabela das distâncias conhecidas até determinado destino, as tabelas são atualizadas por trocas de mensagens entre seus vizinhos. Segundo [Peterson and Davie 2004] também é considerado, na prática, um dos algoritmos mais rápidos de roteamento, faz o uso do conceito matemático do algoritmo de Bellman-Ford para encontrar o caminho mínimo em um grafo ponderado. Routing Information Protocol (RIP) é um protocolo que roda em muitos roteadores na internet, foi projetado para operar em redes intra-domínio de tamanho pequeno e moderado, e usa o algoritmo vetor-distância como base.

2. Decisões de implementação

A primeira decisão tomada no projeto, foi a implementação do roteador, foi definido uma classe chamada Router que implementa todos os métodos que são responsáveis por lidar com as funções específicas do comutador. Todo o tratamento e envio das mensagens é responsabilidade única do roteador, além da inserção e remoção de enlaces vizinhos. Foi criado um dicionário chamado 'listaAdj' para manter o registro dos vizinhos imediatos , também foi criado um outro dicionário para a tabela de roteamento,que explicarei posteriormente. Analisando algumas implementações anteriores me chamou a atenção a criação de threads para lidar com alguns aspectos do programa, por esse motivo adotei em meu código. Foram criadas três daemon threads, a primeira tem como objetivo a remoção de rotas invalidas, após o tempo de renovação de uma rota ser expirado, a thread irá eliminá-la da tabela de encaminhamento e da lista de vizinhos. A segunda tem como função receber as mensagens que chegam e trata-las adequadamente, de acordo com seu tipo (update, trace ou data). Existe também uma última thread, responsável por enviar as mensagens de update no tempo determinado. A implementação mais desafiadora, no meu caso, foi a criação do código para lidar com a tabela de roteamento e o algoritmo de soma de custos. Para facilitar e ficar mais claro foi criado uma classe para a tabela de roteamento, ela é responsável por todas as funções de alteração da tabela além de executar o trecho de código do cálculo de custos do vetor de distâncias. A tabela foi definida como sendo um dicionário que contém outros dicionários(destino: next-hop,custo). Outra questão desafiadora foi entender como teria que ser codificado o ambiente de entrada, creio que a especificação deixa a desejar neste sentido, já que não apresenta exemplos adequados.

Três métodos criados são de grande importância para o algoritmo, são eles: h-update(), responsável por tratar as mensagens de update; h-trace(), têm a mesma função mas trata as mesagens de trace recebidas adicionando seu Ip ao campo hops, caso não seja o Ip de destino; h-data(), trata a mensagens de dados e imprime o conteúdo do pacote.

3. Código comentado

3.1. Atualizações periódicas

Como já foi dito as atualizações periódicas são realizadas por uma daemon thread que é chamada no método Run. A partir daí dois procedimentos ficam resposáveis pela execução.

Listing 1. Resposável por enviar a msg de update de tempos em tempos

```
def send_message_update(self):
    while True:
            time.sleep(self.period)
                self.send_update()
```

Listing 2. Resposável por enviar a msg de update

```
def send_update(self):
    '''Envia a tabela de rotas para roteadores vizinhos '''
    for i in self.listAdj:
            msg = self.msg_update(self.ip, i, self.table.table)
            self.router.sendto(msg, (i,self.port))
```

3.2. Split horizon

De acordo com [Kurose and Ross 2010] Split horizon é uma característica dos roteadores para evitar contagem ao infinito, o procedimento responsável por isso neste código foi o h-update(). Basicamente ele não envia novamente a rota recebida de um determinado vizinho, se caso o roteador que está tratando a mensagem esteja incluído no caminho (next-hop), ele não atualiza sua tabela de roteamento.

Listing 3. Resposável por lidar e tratar a msg de update

```
def h_update(self,msg):
    '''Trata as mensagens de update, adiciona as rotas inexistentes na
       tabela de roteamento e atualiza as rotas '''

  for destination in msg['distances']:
    for next_hop in msg['distances'][destination]:
      if next_hop != self.ip and destination != self.ip:
        cost = int(msg['distances'][destination][next_hop]) + int(self.
            get_costs(msg['source']))
        self.table.add_table(destination,msg['source'],cost)
          self.up_valid(msg['source'])
```

3.3. Reroteamento imediato

O reroteamento imediato é realizado para melhorar a eficiência de um roteador, quando uma rota sai da tabela de rotemaneto ele automaticamente identifica e recalcula o ”novo caminho”. Neste projeto foi implementado da seguinte maneira, antes de enviar cada mensagem de dados ou trace o comutador simplesmente executa o algoritmo de calculo do vetor de distância.

Listing 4. Resposável por lidar e tratar a msg de trace

```
def h_trace(self, msg):
    '''Trata a mensagem de trace, caso seja o destino ele envia
       novamente a msg para fonte como data.Caso contrario ele
       adiciona
    a seu ip e encaminha a msg '''

    '''
      list_next-hops: dict - lista que recebe os hops para onde a msg
         sera encaminhada
    '''

    list_next_hops = self.table.distance_vector_algorithm()
    if msg['destination'] != self.ip:
      msg['hops'].append(self.ip)
      if msg['destination'] in list_next_hops:
        n = msg['destination']
        self.send_Message(msg,list_next_hops[n])
    else:
      payload = json.dumps(msg)
      new_msg = self.msg_data(self.ip, msg['source'], payload)
      n = msg['source']
      self.send_Message(new_msg, list_next_hops[n])
```

Listing 5. Resposável por lidar e tratar a msg de data

```
def h_data(self,msg):
    '''Trata a msg de dados,caso self.ip for o destino a funcao imprime
       o payload caso contrario encaminha a msg '''
    if msg['destination'] == self.ip:
      msg = json.loads(msg)
      payload = '{"type": "' + msg['type'] + '", '
      payload += '"source": "' + msg['source'] + '", '
      payload += '"destination": "' + msg['destination'] + '", '
      payload += '"hops": ' + str(msg['hops']).replace('\'', '"') + '}'
      print(payload)
    else:
      list_next_hops = self.table.distance_vector_algorithm()
      if msg['destination'] in list_next_hops:
        n = msg['destination']
        self.send_Message(msg, list_next_hops[n])
```

3.4. Remoção de rotas desatualizadas

Como já foi dito a remoção de rotas é realizada por uma daemon thread. Basicamente ela verifica os vizinhos válidos, aqueles que estão conectados tem seu valor na lista de válidos setado para zero caso contrário é somado + 1, depois de 4 períodos a rota é retirada dos nexts-hops da tabela de encaminhamento do roteador. A atualização das outras tabelas irá acontecer na medida que a mensagem de update é propagada.

Listing 6. Resposável por remover rotas invalidas

```
def remove_invalid_routes(self):
    '''Responsavel por remover as rotas que ja nao estao ativas da
       tabela de roteamento '''

    while True:
      invalid = []
      time.sleep(self.period)
      for i in self.listValid:
        if self.listValid[i] == 4:
          invalid.append(i)
      for d in invalid:
        if d in self.listValid:
          valid_lock.acquire()
          del self.listValid[d]
          valid_lock.release()
          self.table.remove_table(d)
      valid_lock.acquire()
      for i in self.listValid:
        self.listValid[i] = self.listValid[i] + 1
      valid_lock.release()
```

3.5. Balanceamaneto de carga

O trecho de código responsável pelo balanceamento de carga é o algoritmo de vetor-distância, ele salva a rota de menor custo, todas as rotas com custos iguais são potencias nexts-hop.

Listing 7. Resposável por tratar o balanceamento e determinar o próximo destino

```
def distance_vector_algorithm(self):
    '''Trecho de codigo que faz o processo de identificacao de menor
       rota para cada destino '''

    next_hop = {}
    potencial_min = [] #Balannceamento de carga
    for v,d in list(self.table.items()):
      min_cost = 2**30
      for i in list(d.items()):
        if float(i[1]) < min_cost:
          min_cost = float((i[1]))

      for i in list(d.items()):
        if float(i[1]) == min_cost:
          min_nexthop = i[0]
          potencial_min.append(min_nexthop)
          destination = v

      min_nexthop = random.choice(potencial_min)
      next_hop[destination] = min_nexthop

    return next_hop
```

4. Conclusão

Apesar da complexidade do TP foi interessante implementá-lo, pois adquiri um conhecimento mais profundo sobre protocolos de roteamento e o funcionamento de um roteador. As otimizações estabelecidas pelo professor, me fizeram perceber o quanto é importante a alta capacidade de processamento de um comutador no núcleo da rede.

References

Kurose, J. F. and Ross, K. W. (2010). *Redes de Computadores e a Internet: Uma abordagem top-down*. Pearson, São Paulo, trad. 5 ed. edition.

Peterson, L. L. and Davie, B. S. (2004). *Redes de Computadores: Uma abordagem de sistemas*. Campus, Rio de Janeiro, trad. 3 ed. edition.