Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

TP0

Breno Poggiali de Sousa
brenopoggiali@gmail.com
Vinicius Julião Ramos
viniciusjuliao@dcc.ufmg.br

2 de Novembro de 2020

Resumo

O trabalho aqui desenvolvido consiste na construção de um modelo virtual para um roteador que implementar o protocolo RIP. Nessa aplicação, o algoritmo de *Distance Vector* foi introduzido e através de um código encapsulado com a função apenas de gerênciar a tabela de roteamento. Esse encapsulamento, permitiu que cada uma das funções solicitadas pela especificação do trabalho fossem desenvolvidas de forma independente. Tais funções são: Atualizações periódicas, *Split Horizon*, rerroteamento imediato e a remoção de rotas desatualizadas.

1 Introdução

Para possibilitar a virtualização de um roteador, através de um aplicação, necessitou-se utilizar um protocolo da camada de transporte. Nesse cenário, optou-se pelo UDP, uma vez que esse protocolo é o que mais se aproxima da camada de rede, pelo motivo de não fornecer quaisquer garantias de entrega, controle de erros ou gerenciamento de conexão. Portanto, para esse socket, uma classe **Router** foi criada com a finalidade de gerenciar a troca de informações. Essa mesma classe é responsável por executar comandos em duas *thread* diferentes, sendo que uma delas recebe comandos da entrada padrão do sistema e a outra executa o protocolo de rede especificado pelo trabalho.

A opção de criar uma classe separada para gerenciar a tabela de roteamento permitiu gerênciar todas a funções que cabem à uma tabela de roteamento: a remoção de rotas desatualizadas, a montagem da lista de distâncias através do *Split Horizon* e também o rerroteamente imediato. Essa última função é desempenhada por demanda, o que permite que a melhor rota para determinado endereço seja calculada apenas quando o roteador envia mensagem para tal destino. Já a atualização periódica de rotas é uma atividade conjunta entre o roteador e a tabela de roteamento; a tabela de roteamento entrega ao roteador a lista de distâncias, mas cabe ao roteador atualizar os vizinhos de acordo com o período especificado.

2 Estrutura do código

3 Decisões de Implementação

Essa seção trata de como as quatro principais funções, requeridas pela descrição do trabalho, foram desenvolvidas. Também é colocado em cheque as dificuldades referentes às soluções propostas. Mas antes de descrever a implementação, é necessário abordar três estruturas de dados presentes na classe RouterList:

- **Route**: Atributo **ipaddr**: é o endereço final da rota. Atributo **last_update**: é um atributo que salva a ultima vez em que a rota foi atualizada. Atributo **hops**: é um dicionário **(K_addr, V)** em que **K_addr** é o endereço ip do link associado ao roteador e **V** é o custo de enviar uma mensagem para o endereço **ipaddr** passando pelo link **K_addr**.
- **RouterList.__routes**: Trata-se de um dicionário **(R_addr, R)**. Nesse caso **R_addr** é o endereço de uma máquina e **R** trata-se da instância de **Route** correspondente a tal destino.
- **RouterList.__links**: Dicionário **(L_addr, L)** em que os links de endereço **L_addr** são informados na entrada padrão, através do comando `add`, juntamente com o seu peso. Essa informação é armazenada em uma instância da classe **Link**.

3.1 Atualizações Periódicas

A cada meio segundo, é feito uma chamada ao método `Router.__share_router_list`. Então esse método trata de analisar se já passou tempo suficiente desde a última atualização, para que uma nova atualização seja feita. Caso seja necessário atualizar novamente, então as listas de distâncias são requisitados da tabela de roteamento. Como essa lista de distâncias pode variar de acordo com o link, devido ao *Split Horizon*, o retorno dessa chamada é um dicionário, em que cada elemento **(addr, dist)** corresponde à chave **addr** (endereço) do link e **dist** é a respectiva lista de distancias.

O método itera sobre o dicionário e envia uma mensagem de `update` contendo a respectiva lista de distâncias de cada um dos vizinhos. Observe que o método recebe um atributo **past**, que é um `datetime` da ultima vez em que uma atualização foi enviada. Caso uma atualização seja feita, então retorno `True` informa que o atributo **past** deve ser atualizado para o momento corrente.

```
1  class Router(Thread):
2    {...}
3    def __share_router_list(self, past):
4      now = datetime.now()
5
6      if (now - past).seconds > self.__period:
7        self.__router_list.refresh_routes()
8        all_links_distances = self.__router_list.distances_dictionary()
9
10       for link, distances in all_links_distances.items():
11       msg = Update(self.__ipaddr, link, distances)
12       self.__send_message_as_json(msg)
13
14       return True
15     return False
```

Listing 1: Envio das atualizações periódicas das listas de roteamento

3.2 Remoção de rotas desatualizadas

Essa funcionalidade é desempenhada pela classe **RouterList**. Como tal classe é a única que possui o controle para inserção, remoção e atualização das rotas, é natural que esses trabalho seja executado pela tabela de roteamento. Ao instanciar um objeto **RouterList**, deve-se informar qual o tempo de vida de uma rota desatualizada. Isso serve para que cada rota criada, tenha um *timer* associado, e esse contador de tempo consiga analisar se o tempo de vida foi extrapolado ou não. Observe que objetos **Route** implementa seu próprio *timer* pelo atributo **last_update**.

O período de remoção das rotas desatualizadas é igual a $4 \times period$, em que $period$ é o tempo de atualização das rotas dos links vizinhos. Como no método `__share_router_list` há um controle sobre $period$, optou-se por implementar uma chamada ao método `RouterList.refresh_routes` no mesmo controlador do envio das listas de distâncias; assim como mostra a Linha 7 do Listing 1. Sendo assim, a cada $period$ segundos, valisa-se se há rotas desatualizadas. Isso se dá pelo seguinte código:

```
1  class RouterList:
2    {...}
3    def refresh_routes(self):
```

```
4     deletion_list = []
5     for ipaddr, route in self.__routes.items():
6       if ipaddr not in self.__links:
7         if route.updated_since > self.__max_life:
8           deletion_list.append(ipaddr)
9
10    for ipaddr in deletion_list:
11      del self.__routes[ipaddr]
```

Listing 2: Remoção das rotas desatualizadas

Há outro fator que merece atenção: Quando uma mensagem do tipo `update` chega até o roteador, então a tabela de rotas deve ser atualizada. Nessa atualização da tabela de rotas, todas aquelas rotas contidas na mensagem têm seu *timer* atualizado para o momento corrente. Isso faz com que essa rota ganhe mais tempo de vida.

3.3 Rerroteamento imediato

É importante ressaltar que todo roteamento executado pelo servidor é feito de forma imediata. Ou seja, a melhor rota para determinado destino é calculada apenas quando uma mensagem for enviada para tal endereço. A troca de mensagens é desempenhada pela classe **Router**, sendo que qualquer envio de dados é executado pelo método `Router.__send_message_as_json`. Observando o Listing 3, a cadeia de chamadas que retorna a melhor rota inicia-se na Linha 16, em que o endereço de destino é informado para `Router.next_hop`. Nesse ultimo método, caso não haja uma rota para tal endereço, então um valor `None` é retornado, mas caso contrário, requisita-se o cálculo da rota para classe **Route**. Por fim, `Route.route` retorna o endereço da rota que possui menor peso.

```
1  class Route:
2    {...}
3    def route(self):
4      return min(self.__hops.items(), key=lambda x: x[1])
5
6  class RouteList:
7    {...}
8    def next_hop(self, ipaddress_):
9      if ipaddress_ in self.__routes:
10       return self.__routes[ipaddress_].route[0]
11     return None
12
13 class Router(Thread):
14   {...}
15   def __send_message_as_json(self, message):
16     ipaddr = self.__router_list.next_hop(message.destination)
17     if ipaddr is not None:
18       self.__sock.sendto(str(message).encode(), (ipaddr, DEFAULT_PORT))
```

Listing 3: Retorna a melhor rota

Na abordagem aplicada para esse calculo de rerroteamento mesmo quando uma rota é removida ou alterada da tabela de roteamento, as demais rotas permanecem intactas. Logo, para o caso em que a melhor rota para um endereço X fora removida, quando **Router** desejar enviar outra mensagem para X, uma nova melhor rota será obtida pela chamada de `RouterList.next_hop(X)`

3.4 Spilt Horizon

O método *Split Horizon* é aplicado na geração dos dicionários *distances* que posteriormente serão adicionados às mensagens. Antes de gerar o dicionário **distances** que associa o endereço do vizinho à sua lista de distâncias, gera-se um dicionário contendo todas as melhores rotas (**all_distances**). Então, para gerar **distances**[**link**] – que é o dicionário de distâncias de determinado link – itera-se sobre todos os vizinhos, de forma que **distances**[**link**] recebe todas as rotas que obedecem à optimização *Split Horizon*. Ou seja, **distences**[**link**] recebe um dicionário contendo as rotas as quais o endereço de **link** não seja dado como o melhor caminho.

```
class RouterList:
  {...}
  def distances_dictionary(self):
    all_distances = dict()
    for ipaddr, r in self.__routes.items():
      all_distances[ipaddr] = r.route

    distances = dict()
    for link, linkv in self.__links.items():
      distance = dict()
      for ipaddr, route in all_distances.items():
        if ipaddr != link and route[0] != link:
          distance[ipaddr] = route[1]
      distance[self.__local_ip] = linkv.weight
      distances[link] = distance

    return distances
```

Listing 4: Retorna o dicionário de distâncias usando Split Horizon

4 Diferença Semântica

Cada um dos programas cliente possui uma forma particular de comunicação, um protocolo, na qual, de acordo com a senha recebida, o servidor destina o fluxo de execução para uma comunicação entre o cliente do professor ou para o cliente do aluno; no caso de não receber alguma senha válida, a conexão é encerrada. Algumas mensagens trocadas entre os programas são padrões do protocolo, e para estes casos, ao receber uma mensagem padrão quaisquer dos programas verificam se o conteúdo do pacote recebido é realmente o que se espera. Entretanto, há algumas mensagens que têm conteúdo variável, para estes casos é necessário testar se a quantidade de bytes recebida é o que se espera, ou então aguarda-se alguma mensagem para o fim da comunicação.

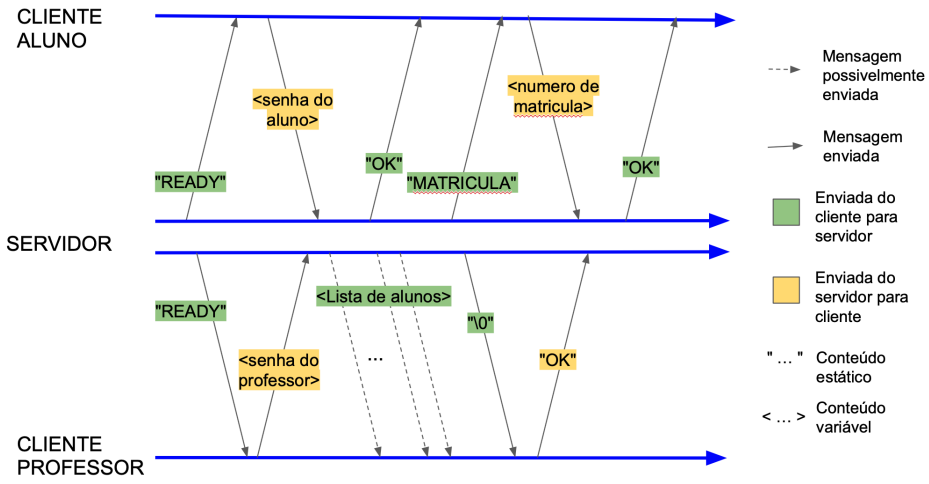

Figura 1: Protocolo de comunicação entre clientes e o servidor.

O diagrama da Figura 1 exibe como se dá a forma de comunicação entre os programas em que há dois detalhes que merecem certa atenção: na troca de mensagens do fluxo do professor, no caso deste trabalho, implementou-se de forma que o servidor envia uma *string* contendo um número de matricula seguido por um caractere **\n** em cada envio, ao final do envio da lista há o envio do caractere **\0**, identificando o fim da lista. O fechamento da conexão é realizado após o envio de uma mensagem "OK"pelo cliente. Vale lembrar que o maior inteiro de trinta e dois bits (quatro bytes) é o número 4294967295, que possui dez caracteres em sua representação *string*, logo, são enviados de onze em onze bytes, no máximo, sendo que tal mensagem

tem tamanho variável, dependendo da quantidade de bytes necessária para representar tal inteiro de forma decimal.

No segundo caso, na troca de mensagens entre o cliente aluno a informação contendo o número de matrícula do aluno possui conteúdo variável, mas o tamanho é fixo de quatro bytes, o que representa um inteiro de trinta e dois bits. O tamanho de tal mensagem é validado sempre que recebida, e computa-se o número de matrícula apenas após o recebimento dos quatro bytes da matrícula, seguidos do envio com sucesso da mensagem "OK"pelo servidor; sendo que essa ultima mensagem identifica o final da comunicação.

5 Tratamento de múltiplas conexões

As múltiplas conexões foram tratadas de duas maneiras diferentes, e para isso desenvolveu-se dois tipos de arquivo servidor diferentes; um que utiliza de *threads* e outro não. No caso do servidor que não utiliza *threads* (arquivo **servidor.c**), as múltiplas conexões são tratadas de forma que se permite dez conexões simultâneas, entretanto, o servidor responde apenas uma por vez de modo que após receber uma primeira comunicação com determinado cliente, a segunda conexão só iniciará a troca de mensagens após a finalização da primeira, criando uma fila de prioridades na qual a função *listen()* recebe como parâmetro a quantidade de elementos (conexões) que essa lista terá.

Quanto ao caso do servidor multi processos (arquivo **servidor-mt**) a função *listen()* também implementa dez conexões para a fila de prioridades, porém ao utilizar a execução paralela de conexões, em um *poll* de tamanho cinco, pode-se comunicar simultaneamente com cinco clientes enquanto outros cinco aguardam na fila de prioridades, até que haja uma "vaga"no *poll* e assim serão executados.

6 Implementação e Execução

Dadas as especificações do trabalho, necessitou-se organizar o código e implementar funcionalidades que viabilizassem a boa execução dos programas. Dentre tais decisões destaca-se o uso de um *array* com duzentos e cinquenta e seis posições para armazenar os alunos, ou seja, o número máximo de alunos que podem ser registrados é 256. Esse número foi acertado com o professor em sala de aula para que os alunos se prevenissem de implementar a estrutura de lista encadeada. Quanto à geração automática das senhas de acesso ao servidor, as senhas são compostas de algarismos alpha numéricos, valendo destacar que as senhas são *case sensitive* das quais a primeira cadeia de 8 bytes impressa é a senha do professor e a última é a senha dos alunos.

Já na implementação do **cliente-aluno**, acrescentou-se um novo parâmetro de execução que representa o número de matrícula do aluno, dada a não especificação na descrição do trabalho, optou-se por permitir que cada aluno realize a chamada passando seu próprio número de matrícula. Caso tal parâmetro não seja passado, ou algum outro parâmetro, então o programa retornará uma mensagem de erro. Essa condição dos parâmetros é válida para todos os programas desenvolvidos.

Da **execução** dos programas, antes de qualquer chamada dos programas, deve-se executar o comando **make**, e então quatro arquivos executáveis serão gerados (**servidor**, **servidor-mt**, **cliente-professor**, **cliente-aluno**), sendo um destes o arquivo *multi threads* possuindo o sufixo "mt". a chamada desses aquivos se dará da forma como descria na documentação do trabalho, mas vale destacar que a chamada de qualquer um dos arquivos servidor resultará na impressão de duas senhas, a primeira representando a senha do professor e a segunda a senha dos alunos.

Sobre a execução do programa **servidor-mt**, essa se dará da seguinte forma:

```
$ ./servidor-mt <porta tcp>

```

na sequência do comando acima haverá a impressão das senhas, como já especificado anteriormente.

Quanto à execução do programa **cliente-aluno**, basta adicionar um ultimo parâmetro comparando-se com a forma de execução solicitada pela descrição do trabalho:

```
$ ./cliente-aluno <ip servidor> <porta tcp servidor> <senha aluno>
<matricula aluno>

```

Quanto às demais execuções, segue-se da maneira que foi solicitada pela descrição do trabalho.

7 Conclusão

O trabalho prático foi útil para visualizar a comunicação TCP entre clientes e servidor utilizando dois tipos diferentes de tratamento de múltiplas conexões, além de desenvolver uma aplicação para ser utilizada em rede, que exigiu uma implementação muito bem organizada das funções e procedimentos.

Com uma pequena execução de duzentos e cinquenta chamadas em paralelo da aplicação cliente, primeiro para o servidor multi tarefas e na sequencia para o servidor com um tratamento mais básico de conexões, pôde-se observar que a quantidade de comunicações sem sucesso foi muito inferior para o caso do servidor que utiliza *threads*. Tal fato se explica pela liberação de novas conexões mais rapidamente do que para o caso em que a execução é sequencial e são tratadas apenas um cliente por vez.