Trabalho Prático 2 de Redes

DCCRIP

Douglas Rodrigues de Almeida
douglasralmeida@live.com

1. Introdução

O objetivo deste trabalho é implementar um simulador de roteamento de redes. A simulação deve incluir um roteador que utiliza roteamento por vetor de distâncias com suporte a pesos nos enlaces, balanceamento de carga e medição de rotas.

O roteador inclui um interpretador de linhas de comando básico onde o usuário pode incluir e excluir distâncias manualmente na tabela de distâncias, traçar rotas e exibir a tabela de distâncias de qualquer roteador da rede.

2. Desafios e Soluções adotadas

A necessidade de implementar um interpretador de comandos dificultou a implementação do soquete que recebe os dados da rede pois a escuta bloqueia a thread principal do aplicativo prejudicando a digitação de comandos no interpretador. A solução foi criar uma aplicação multi-thread onde alguns procedimentos são executados em threads separadas. Foi implementada uma thread para escuta da rede e outra thread para envio de dados e uma thread para consumir os dados recebidos.

A solução multi-thread trouxe outros desafios que tiveram que ser solucionados. A thread de escuta da rede executando sequencialmente estava utilizando muitos recursos da CPU, ainda que não houvesse dados para receber. Para corrigir este problema, foi implementando um timeout de 1 segundo na função de recebimento de dados. Assim, a cada segundo, se não houver dados para receber, a execução da thread de recebimento não ficará bloqueada.

Outro problema apresentado pela solução multi-thread foi a ausência de sincronização entre as threads geradoras e consumidoras de dados e as threads de envio e recebimento de dados na rede. Para solucionar este problema, foram implementadas filas de entrada e saída de mensagens usando o tipo thread-safe Queue.

Por fim, encontrei um problema na especificação das mensagens JSON e a biblioteca padrão de manipulação JSON do Python. De acordo com a especificação, o campo *payload* que envia a tabela pela rede quando solicitada pelo comando *table* deve conter as rotas em tuplas que, em Pyhton, são representadas por parênteses. Entretanto, a biblioteca JSON converte as tuplas em listas pois não existem tuplas na especificação JSON. Considerando que este comportamento está previsto na especificação, ele não foi alterado. As tuplas de rotas são encaminhadas pela rede como listas.

3. Funcionalidades implementadas

3.1 Atualizações Periódicas

Cada roteador da rede simulada envia sua lista de distâncias periodicamente para seus vizinhos. Isto é feita na thread de RotasAtualizadas. Ela funciona sob um laço infinito que gera a lista de distâncias conhecidas, as envia aos vizinhos e depois dorme por um período. Seu código tem a seguinte estrutura:

```
while (thread.ativa):
  vizinhos = Enlaces.obtertudo()
  for vizinho in vizinhos:
    distancias = DistanciasConhecidas.obter(vizinho)
```

```
            enviar(vizinho, distancias)
        time.sleep(intervalo)
```

3.2 Split Horizon

A lista de distâncias conhecidas possui uma otimização conhecida como split horizont. As distâncias enviadas ao vizinho *x* não possui rotas para *x* nem rotas aprendidas de *x*. Rotas aprendidas de *x* são rotas cujo próximo passo é *x*. Essa eliminação é feita durante a construção da lista de distâncias que será encaminhada a um vizinho:

```
for (ip, distancia) in DistanciasConhecidas:
  if vizinho == ip or vizinho == distancia.prox:
    continue
  ListaOtimizada.Add(ip, distancia)
```

3.3 Balanceamento de Carga

Os roteadores são capazes de balancear a carga de dados transmitidos na rede. Para tal, é armazenada uma lista de todas as rotas de mesmo peso para um destino específico. Ao transmitir um pacote, uma rota é escolhida aleatoriamente da lista. Assim, garante-se uma transmissão uniforme entre as rotas.

Código para escolher uma rota:

```
for (ip, distancia) in DistanciasConhecidas:
  if ip == destino:
    rota = distancia.prox[randrange(distancia.quantidade)]
```

3.4 Reroteamento Imediato

Quando uma rota deixa de existir na rede, um roteador será capaz de substituí-la imediatamente para outra rota conhecida, caso exista uma. Isso é possível porque ele armazena todas as rotas conhecidas para um destino e não apenas a melhor. As rotas são armazenadas numa lista ordenada. Havendo exclusão de uma, a próxima da lista será utilizada.

3.5 Remoção de Rotas Desatualizadas

Cada rota armazenada no roteador simulado possui um tempo de vida representada pelo campo *tempovida*. A cada envio de rotas atualizadas para os vizinhos, as rotas diminuem um ponto no tempo de vida. Quando o tempo de vida alcança zero, a rota é eliminada. Sempre que o roteador recebe uma informação atualizada da rota pela, seu tempo de vida é restaurado.

3.6 Mensagens de Erro

Os roteadores simulados são capazes de enviar mensagens de erro a origem quando uma mensagem *table* ou *trace* não encontra uma rota até o seu destino. Uma mensagem do tipo *data* é enviada a origem contendo no campo *payload* a mensagem de erro avisando sobre a rota não encontrada. Pela definição do trabalho, quando uma mensagem do tipo *data* é recebida pelo destino seu campo *payload* será exibido na tela, informando o usuário do erro ocorrido.