Redes De Computadores - Trabalho Prático 2

Alunos: Renato Sérgio Lopes Júnior - 2016006875
Tiago Negrisoli de Oliveira - 2016006956

1 Introdução

Neste trabalho foi desenvolvido um sistema de roteadores que consistem em *x* roteadores. Estes podem fazer conexões entre si para que seja possível o envio de mensagens. Cada roteador guarda uma lista de quem ele se conecta (também chamado de vizinho). De forma automática e a cada período de tempo *t* , os roteadores compartilham com seus vizinhos essa lista. Assim , cada roteador vai aprender qual o mapeamento da rede que se encontra e em qual enlace deve enviar uma mensagem *m* para que chegue a um destinatário *d* , este sendo seu vizinho (envio direto pelo enlace para o destino) ou não (envio para outro roteador encaminhar para o destino).

2 Desenvolvimento

A implementação do trabalho foi feita na linguagem java. Para realizar as conexões entre roteadores, foi utilizada a classe java.net.DatagramSocket.

Para se fazer os roteamentos, foi criada a classe *RouterRIP*, que representa os roteadores. Está contido nela a tabela de rotas (que guarda as menores rotas para um determinado destino e para onde tem que enviar a mensagem) e as funções de receber, tratar e enviar mensagens, estas podendo ser de dados, roteamento ou de *trace*.

Para lidar com a parte de enviar mensagens de roteamento para atualizar as tabelas de rotas a cada x segundos, receber comandos do usuário e o roteador ficar esperando uma mensagem chegar sem que fique travado, foram utilizadas threads. Dessa forma, todas essas funções funcionam independentemente.

Embora a documentação exigisse que todo o código fonte fosse feito em um único arquivo, devido ao extenso tamanho, foi dividido em vários para facilitar o desenvolvimento e a compreensão de leitores.

3 Implementação das funcionalidades

- Atualizações periódicas
    - Foi criada a thread `UpdateRoutesThread`. Nela, são enviadas as mensagens de update para os roteadores vizinhos. Para isso, foi criado o método *`sendUpdateToNeighbour`*, que envia as mensagens de update para os vizinhos. Para verificar quais roteadores são vizinhos, foi criado o método *`verifyNeighbour`*, que retorna true caso para um dado roteador *R* tenha uma rota cujo next hop seja ele mesmo, o que indica que R é um roteador vizinho.

```java
/**
 * Verifica se o roteador é vizinho de router. Roteador será vizinho se o ip
 * de destino e o next hop forem iguais na entrada da tabela de roteamento.
 *
 * @param r o possível vizinho.
 * @return true, se for vizinho, e falso, se não for.
 */
private synchronized boolean verifyNeighbour(RoutingTableEntry r) {
    return r.getIpDestination().equals(r.getNextHop());
}
```

- *Split Horizon*
    - Para implementar o *split horizon*, foi alterado o método *sendUpdateToNeighbour*, discutido no tópico anterior. Antes de enviar a rota r para o roteador R, é verificado se o next hop é diferente do ip do vizinho para o qual está sendo enviada a mensagem de update e se o destino da rota r também é diferente desse ip do vizinho. No código, isso ficou da seguinte forma:

```java
for (RoutingTableEntry r : router.getKnownRoutes()) {
    if (!r.getNextHop().equals(neighbourIp) && !r.getIpDestination().equals(neighbourIp)) {
        updateMessage.addDistance(r.getIpDestination(), r.getDistance());
    }
}
```

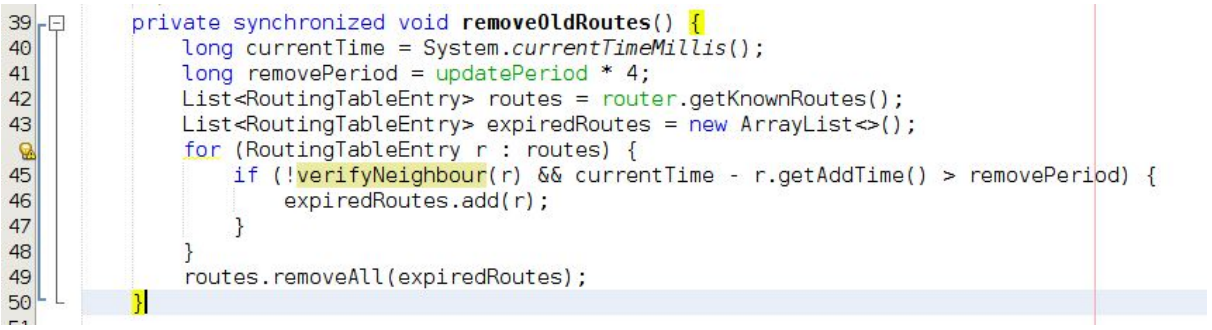
- Balanceamento de Carga
    - Para implementar o balanceamento de carga, foi alterado o método de escolha da melhor rota para dado destino. Caso tenha mais de uma rota, é escolhida uma aleatoriamente usando a classe Random de java.util.

```java
private synchronized RoutingTableEntry getBestRouteToDestination(String ipDest) {
    ArrayList<RoutingTableEntry> bestRoutes = new ArrayList<>();
    int bestDistance = -1;
    // Pega a menor distância
    for (RoutingTableEntry r : this.knownRoutes) {
        if (r.getIpDestination().equals(ipDest)) {
            if (bestDistance == -1) {
                bestDistance = r.getDistance();
            } else if (r.getDistance() < bestDistance) {
                bestDistance = r.getDistance();
            }
        }
    }
    for (RoutingTableEntry r : this.knownRoutes) {
        if (r.getIpDestination().equals(ipDest) && r.getDistance() == bestDistance) {
            bestRoutes.add(r);
        }
    }
    if (bestRoutes.isEmpty()) {
        return null;
    }
    return bestRoutes.get(new Random().nextInt(bestRoutes.size()));
}
```

- Rerroteamento Imediato
    - Para implementar o rerroteamento imediato, foi alterado o método de salvamento de rotas recebidas. Em vez do roteador guardar apenas as rotas mais curtas, ele guarda todas as rotas que receber para dado destino. Assim, caso as mais curtas deixem de existir, ele ainda tem outras opções de rota.

```java
case "update":
    if (messageJson.getString("destination").equals(this.ip)) {
        Map<String, Object> routesReceived = messageJson.getJSONObject("distances").toMap();
        for (String key : routesReceived.keySet()) { // Adiciona todas as rotas que receber do update
            addNewRoute(key, messageJson.getString("source"), (Integer) routesReceived.get(key));
        }
    }
    break;
```

- Remoção de Rotas Desatualizadas
    - Para implementar a remoção de rotas desatualizadas, em cada rota que é salva na tabela de roteamento, é salvo o instante, em que ela foi adicionada. Na thread UpdateRoutesThread, foi adicionado o método removeOldRoutes, que verifica quais rotas foram adicionadas a mais de 4*Π segundos. Essas rotas vencidas são, então, removidas da tabela de roteamento do roteador.



```
39      private synchronized void removeOldRoutes() {
40          long currentTime = System.currentTimeMillis();
41          long removePeriod = updatePeriod * 4;
42          List<RoutingTableEntry> routes = router.getKnownRoutes();
43          List<RoutingTableEntry> expiredRoutes = new ArrayList<>();
            for (RoutingTableEntry r : routes) {
45              if (!verifyNeighbour(r) && currentTime - r.getAddTime() > removePeriod) {
46                  expiredRoutes.add(r);
47              }
48          }
49          routes.removeAll(expiredRoutes);
50      }
```

4 Instruções de compilação e execução

O trabalho foi feito utilizando a IDE NetBeans, entretanto foi feito um Makefile para a compilação ser facilitada sem a utilização da IDE. Para compilar e rodar o trabalho, basta seguir as instruções:

1. Abrir o terminal na pasta src
2. Executar o seguinte comando no terminal:
   `$ make`
3. Após a compilação, para rodar basta executar no terminal o comando:

```
$ java  -jar  router.jar  ENDERECO_IP  PERIODO
```

Ou

```
$ java  -jar  router.jar  ENDERECO_IP  PERIODO  ARQUIVO_INICIALIZACAO
```

Onde ENDERECO_IP é o endereço IP que será usado pelo roteador, PERIODO é o período de atualização de rotas, em segundos, e ARQUIVO_INICIALIZACAO é o arquivo com comandos para ser executados durante a inicialização do roteador.

5 Conclusão

A partir da implementação deste trabalho, foi possível ver o funcionamento de um sistema automático de redes e entender melhor as dificuldades de implementação de um sistema mais robusto como esse, em contrapartida com as vantagens dele. Em uma rede de escala grande, sua forma dinâmica de atualizar o tráfego de dados e a manutenção da mesma torna esse método imprescindível.