Implementação de um Serviço de Notícias numa Rede Adhoc

Nuno Areal(A74714), Mário Silva(A75654)

Arquiteturas Emergentes de Redes - MiEI
Universidade do Minho

Abstract. Neste relatório explicamos de que forma é que realizamos as diferentes partes do trabalho prático 1. Explicaremos quais os protocolos que utilizamos, de que forma os utilizamos, como foram projetados e de que forma são uma boa solução para a rede Adhoc.

1 Introdução

O primeiro trabalho prático baseia-se na implementação de um protótipo de um serviço de notícias numa rede Adhoc através de um protocolo de encaminhamento e de um protocolo aplicacional. Esta implementação supõe as seguintes funcionalidades:

- Cada nó conhece os seus vizinhos através do envio de receção de mensagens HELLO por UDP no endereço FF02::1 em *multicast* na porta 9999.
- Nas mensagens HELLO será também enviado os seus vizinhos diretos. Desta forma todos os nós conhecerão os seus vizinhos de raio 1 e 2
- Cada nó é capaz de descobrir rotas para outros nós fora da sua vizinhança, de raio maior que 2, através de pedidos ROUTE_REQUEST que procuraram descobrir o nó por *flooding* e retornar uma mensagem de ROUTE_REPLY que percorrerá o caminho inverso, preenchendo devidamente as tabelas de cada nó.
- O serviço de notícias será uma aplicação de utilizador que comunicará através de TCP com a aplicação de encaminhamento, tendo como objetivo retornar noticias através de uma mensagem GET_NEWS_FROM e obtê-las no formato NEWS.

2 Protocolos de encaminhamento

De seguida iremos explicar como estão desenhadas os diferentes PDUs usados para a implementação e manutenção da rede Adhoc. Decidimos utilizar 5 tipos de PDUs, HELLO, ROUTE_REQUEST, ROUTE_REPLY, GET_NEWS_FROM e NEWS.

2.1 Protocolo HELLO

Este protocolo é responsável por manter as tabelas de encaminhamento de cada dispositivo com os vizinhos de nivel 1 e 2, ou seja com um número de saltos igual a 1 e 2, respetivamente. Para tal aplica o que vamos explicar de seguida.

Primitivas de comunicação

- Send:
  - Envia os pacote por *multicast* através de um *socket* UDP na porta 9999
- Receive:
  - Recebe os pacotes na porta 9999 UDP que fazem parte do grupo de *multicast*;
  - Processa o pacote que recebeu, adicionando à sua tabela de encaminhamento os IPs recebidos, tendo como próximo salto o vizinho que os enviou.

Formato das mensagens protocolares As mensagens HELLO estão no formato "HELLO <endereço do vizinho 1><endereço do vizinho 2><...>" (endereço de vizinhos na sua vizinhança de raio igual a 1 separados por espaços).

Interações Para o envio é criada uma *thread* que de x em x tempo envia uma mensagem no formato descrito anteriormente através de *multicast*. No evento da receção de uma mensagem deste tipo é adicionado o endereço do vizinho que lhe enviou e todos os endereços vizinhos que se encontram na mensagem. Todas as mensagens seguintes passam por um teste do estado que compara os vizinhos mencionados na mensagem anterior com a atual, efetuando as modificações devidas na tabela de encaminhamento. Para cada vizinho direto é criada uma *thread* que vai efetuar parte do a processar todas mensagens HELLO desse vizinho, tendo esta um sistema de *timeout* que caso não receba nenhuma mensagem num determinado período elimina este vizinho da tabela assim como todos os vizinhos indiciados por este.

2.2 Protocolo ROUTE_REQUEST

Este protocolo tem a função de permitir a um dispositivo descobrir qual o próximo salto para um endereço na rede que não seja seu vizinho até nível 2. Desse modo desenvolvemos o seguinte para que isso fosse possível.

Primitivas de comunicação

- Send:
  - Envia as mensagens através de *multicast* para os seus vizinhos de nível 1
  - Guarda informação de que existe um ROUTE_REQUEST para o IP destino
- Receive:
  - Ao receber um ROUTE_REQUEST verifica se tem de o reencaminhar ou se pode responder

Formato das mensagens protocolares As mensagens de ROUTE_REQUEST têm o seguinte formato, "ROUTE_REQUEST <IP origem do pedido><IP a descobrir><Nº Saltos><Tempo limite>"

Interações Para fazer o *handle* dos pedidos de ROUTE_REQUEST é criada uma *thread* assim que é recebido um pacote desse tipo. Essa *thread* vai depois verificar se o IP para o qual e necessário descobrir a rota já está na sua tabela de encaminhamento.
Se não estiver e ainda houver saltos a fazer, esta irá adicioná-lo e marcá-lo como sendo um ROUTE_REQUEST e depois enviá-lo para os seus vizinhos. Se estiver na tabela, não estiver marcado como ROUTE_REQUEST e for vizinho de nível menor que 2 esta responde imediatamente. Por último, se estiver na tabela, não estiver marcado como ROUTE_REQUEST, vizinho de nível maior que 2 e ainda houver saltos a fazer verifica se a entrada na tabela ainda está dentro do tempo de validade. Caso esteja responde com um ROUTE_REPLY, caso contrário vai à procura de um caminho através de um ROUTE_REQUEST, e se encontrar responde a quem lhe fez o pedido.

2.3 Protocolo ROUTE_REPLY

Primitivas de comunicação As primitivas de comunicação são as mesmas que as do ROUTE_REQUEST, uma vez que é a mesma *thread* que trata dos dois tipos de PDUs.

Formato das mensagens protocolares As mensagens de ROUTE_REPLY são do seguinte formato, "ROUTE_REPLY <Próximo IP><Nº Saltos><IP a descobrir>"

Interações Ao receber um ROUTE_REPLY a *thread* responsável irá incrementar o numero de saltos, atualizar a informação da sua tabela e enviar o novo ROUTE_REPLY para os seus vizinhos.
A utilização de um Próximo IP na mensagem permite que ao enviar as mensagens em *multicast* apenas o vizinho cujo IP é o indicado continue a fazer o envio por *multicast*. Deste modo iremos eliminar uma quantidade elevada de mensagens em circulação na rede.

2.4 Protocolos GET_NEWS_FROM e NEWS_FROM

Este dois protocolos são responsáveis por fazer o encaminhamento de noticias na rede Adhoc, procedendo da seguinte forma.

Primitivas de comunicação

- Send:
  - Ambas são enviadas por *multicast* na porta 9999
- Receive:
  - No caso do GET_NEWS_FROM verifica se envia por *multicast* ou se responde com NEWS_FOR.
  - Para o NEWS_FOR verifica se envia por *multicast* ou se responde para o socket TCP.

Formato das mensagens protocolares As mensagens de GET_NEWS_FROM são formadas da seguinte forma, "GET_NEWS_FROM <IP origem><IP destino><Próximo IP >".

As mensagens NEWS_FOR têm o seguinte formato, "NEWS_FOR <IP origem ><IP destino ><Próximo IP ><Notícias >".

Interações No momento da receção de mensagens do tipo GET_NEWS_FROM irá ser verificado se o Próximo IP é o do nó que recebeu a mensagem, evitando propagação desnecessária de pacotes e se o IP destino já consta da tabela de encaminhamento. Se este último não acontecer é desencadeado um ROUTE_REQUEST de forma a descobrir o nó destino. Se este for descoberto o envio da mensagem continua por *multicast*, caso contrário é parado o envio.É ainda verificado se o Próximo IP é igual ao IP destino, e caso isso aconteça é enviado por TCP o pedido de GET_NEWS_FROM posteriormente processado pela *thread* responsável. No caso do NEWS_FOR é também verificado o Próximo IP pelas mesmas razões. Como neste o IP destino é já o originador do GET_NEWS_FROM apenas temos de colocar no Próximo IP o endereço do nó vizinho que nos leva lá e enviar a mensagem.
Se o Próximo IP for igual ao IP destino é enviado para o cliente, através de TCP, uma mensagem com "NEWS_FOR <IP origem ><Noticias >".

3 Protocolo de aplicação

Este protocolo é o responsável por fazer uso do que o protocolo de encaminhamento produz, emulando o que seria uma aplicação a correr por cima de um *router*. Através de um modelo cliente/servidor, um nó da rede pode pedir a outro as noticias que este tenha.

3.1 Primitivas de comunicação

- Send:
  - Envia as mensagens para o socket TCP do localhost
- Receive:
  - Recebe as mensagens do socket TCP do localhost

3.2 Formato das mensagens protocolares

Temos dois tipos de mensagens:

- GET_NEWS_FROM
  - "GET_NEWS_FROM <IP origem ><IP destino >"
- NEWS_FOR
  - "NEWS_FOR <IP origem ><Notícia/Dados >".

3.3 Interações

Tudo aqui é muito simples. O cliente procede ao envio por TCP da mensagem com *GET_NEWS_FROM IP_origem IP_destino*. Posteriormente a *thread* já no protocolo de encaminhamento verifica se já existe uma entrada para o IP destino na tabela de encaminhamento. Se esta for encontrada, é feito o envio por UDP do pedido de notícias, caso contrário é feito um ROUTE_REQUEST para encontrar o caminho, que se não for encontrado avisa o cliente com uma mensagem no ecrã. O cliente fica depois a aguardar a receção da mensagem por um certo período de tempo até se dar um timeout.
No caso da receção de um NEWS_FOR pelo cliente significa que as notícias que ele pediu efetivamente chegaram. É então imprimido no ecrã de onde vieram as noticias e o seu conteúdo.

4 Implementação

A próxima parte destina-se a explicar alguns detalhes que não foram referidos até ao momento relacionados com a forma como implementamos a nossa aplicação.

4.1 Detalhes

A implementação foi feita em Java versão 7, usando bibliotecas bem conhecidas. Existe uma grande subdivisão do processamento através de *threads* bem definidas em classes específicas.

A classe **No** contém a especificação da nossa tabela de encaminhamento e contém os parâmetros endereço, endereço do vizinho, saltos até chegar ao endereço, uma blocking queue utilizada na implementação do *dead interval* do protocolo HELLO, e o timestamp para controlo da cache de rotas para vizinhos

de nível superior a 2.

A classe **Adhoc** é a que contém o *main* do nosso programa que implementa o protocolo de encaminhamento, efetua o *start* às varias *threads* necessárias para o funcionamento, nomeadamente a que efetua o envio de hellos, HelloSendThread, a que efetua o processamento de pacotes recebidos por *multicast*, MulticastReceiveThread e a que processa as conexões TCP, TCPThread. Esta também efetua o *print* de uma interface de opções que permite ao utilizador imprimir a tabela de encaminhamento através da *thread* PrintThread.

A classe **HelloSendThread** é uma das *threads* iniciadas pela Adhoc e esta envia as mensagens de HELLO para os vizinhos diretos, incluindo nela os seus de vizinhos diretos. Isto permite que todos os vizinhos diretos conheçam a sua vizinhança num raio de dois saltos.

A classe **MulticastReceiveThread** recebe todos os pacotes que são enviados por *multicast*, efetuando diferentes processos dependendo do tipo da mensagem e do estado atual da tabela de encaminhamento. Os procedimentos mais relevantes são a criação de uma *thread*, HelloReceiveThread, por cada vizinho direto e redirecionamento de todos os pacotes HELLO seguintes para esta thread. Efetua o mesmo processo para os outros pacotes, criando uma RouteThread para pacotes de ROUTE_REQUEST e ROUTE_REPLY e uma NewsThread para GET_NEWS_FROM e NEWS_FROM.

A classe **HelloReceiveThread** tem como função a implementação do parâmetro *dead interval*, que declara o vizinho inatingível caso este não envie nenhuma mensagem de HELLO num determinado período de tempo, eliminando-o da tabela de encaminhamento assim com todos os nós que cuja rota tem como próximo salto esse mesmo vizinho.

A classe **RouteThread** é uma classe que auxilia a MulticastReceiveThread nos pacotes que dizem respeito à descoberta de rotas. Aquando da receção de um desses pacotes, analisa-o e processa-o de acordo com o descrito anteriormente, enviando-o novamente ou descartando o pacote. De referir que aquando do envio de um ROUTE_REQUEST é criada uma *thread*, RequestTimeoutThread, que passado o tempo limite verifica se já chegou a resposta, ROUTE_REPLY. Não chegando remove a entrada criada aquando do envio do *request*.

A classe **NewsThread** também auxilia a MulticastReceiveThread tratando de todos os pacotes relacionados com notícias. O processo do processamento também foi descrito anteriormente e o resultado final é semelhante ao da RouteThread, adicionando apenas a interação TCP.

A classe **TCPThread** é a responsável pelo processamento de pacotes que chegam através de TCP. Esta verifica se se trata de um GET_NEWS_FROM ou de um NEWS_FOR.
No primeiro caso, é verificado se o IP origem consta da tabela de encamin-

hamento. Não estando é efetuado ROUTE_REQUEST para a descoberta do vizinho. De seguida é criado o NEWS_FOR que irá levar as notícias até ao requerente, que será depois enviado por *multicast*.
Na receção de um NEWS_FOR, é recolhido de uma tabela temporária o socket criado com o cliente e são enviadas as notícias.

A classe **PrintThread** efetua o *print* para o ecrã do estado da tabela atual com os parâmetros endereço do nó, endereço do nó vizinho, numero de saltos.

A classe **Cliente** representa o protocolo aplicacional que é independente do programa que implementa o protocolo de encaminhamento. Esta é composta por uma interface que permite efetuar um pedido de GET_NEWS_FROM a um outro nó através do seu endereço IP, criando uma conexão TCP com o programa que esta a efetuar o protocolo de encaminhamento no *host*, enviando este pedido e ficando à espera de uma resposta. Se esta não chegar dentro do tempo limite é escrita no ecrã uma mensagem de erro, avisando que o pedido expirou.

4.2 Parâmetros

- MulticastSocket: 9999 (UDP)
- InetAddress: FF02::1
- ServerSocket: 9999 (TCP)
- Hello Interval: 3 segundos
- Dead Interval: 6 segundos
- Timeout REQUEST: 300 milisegundos, aumentando para os GET_NEWS em 100 ms até um limite de 5 segundos
- Timeout TCP: 5 segundos

4.3 Biliotecas de funções

- MulticastSocket
- InetAddress
- DatagramPacket
- BlockingQueue
- ArrayBlockingQueue
- NetworkInterface
- BufferedReader
- Socket
- InputStreamReader
- DataOutputStream
- PrintWriter
- ServerSocket

5 Testes e Resultados

Em anexo estão vários *prints* de tabelas de encaminhamento para a topologia v6 fornecida na plataforma de e-Learning.

Para executar os testes utilizamos varios nós, mas os *prints* utilizam o A0, A3, A10 e A11.
Podemos encontrar as tabelas de encaminhamento dos 4 nós e também a tabela do nó A11 depois de A0 efetuar um pedido de notícias.
Em cada nó podemos definir a noticia que este tem para dar, encontrando-se também em anexo um imagem que exemplifica isso.
Por fim falta apenas a receção da noticia por parte do A0 e a sua impressão que se encontram também demonstrados numa imagem.

Para além destes verificamos se a movimentação de nós fazia com que houvesse uma alteração nas tabelas, ou seja, se um nó de nível maior do que 2 passava a vizinho direto e de nível 2 e se um GET_NEWS_FROM a um vizinho direto não despoletava um ROUTE_REQUEST.

6 Conclusões e trabalho futuro

Com este trabalho podemos perceber melhor o funcionamento de redes Adhoc e como estas fazem o reencaminhamento de tráfego pela rede e também as diversas formas de otimizar os recursos da rede.
Penso que atingimos os objetivos pretendidos para o trabalho de fazer a implementação de um protocolo de encaminhamento que atenuasse os efeitos de *flooding* típicos de uma rede Adhoc através de várias medidas.
Uma das coisas a mudar seria a separação do servidor da parte do encaminhamento, uma vez que o temos de ter lá para conseguirmos que a nossa aplicação esteja à escuta na porta 9999 em TCP, embora isso não seja muito problemático uma vez que o foco do trabalho prático é o protocolo de encaminhamento. Isso poderia ser conseguido se a tabela de encaminhamento fosse acessível pela parte aplicacional.

Tendo como base o que aprendemos com a realização deste trabalho poderíamos no futuro desenvolver um protocolo de encaminhamento para redes Adhoc mais eficiente, evitando ainda mais os pacotes desnecessários a circular na rede.