ADHOC

Miguel Fernandes 44024
Comunicações por Computador
Engenharia Informática

1 Introdução

Numa era em que as tecnologias móveis assumem particular destaque no nosso dia a dia e a partilha de informação é cada vez maior, torna-se natural que cada vez mais haja a necessidade de mecanismos de estabelecimento de redes espontâneas. Sendo a mobilidade uma das principais características destas redes, o modelo ad-hoc assume particular destaque neste segmento.
Neste projeto pretende-se desenvolver uma aplicação capaz de estabelecer uma rede adhoc, descobrindo os nós da rede com informação proveniente de outros nós e permita ainda a troca de mensagens entre esses nós.
Uma aplicação deste género podia ser bastante interessante para uma empresa de média/grande dimensão permitindo a troca de mensagens dentro de uma determinada área sem qualquer tipo de custos associados a operadoras de telecomunicações.

2 Especificação do protocolo

2.1 Primitivas de comunicação

A rede ad-hoc desenvolvida assenta sobre 4 primitivas que permitem a criação, otimização e envio de mensagens entre as várias entidades. As primitivas implementadas neste trabalho foram as seguintes:

- HELLO – esta primitiva tem como objetivo a descoberta e gestão da lista de nós conhecidos. De referir que esta primitiva tem um campo "tipo" que permite definir se uma determinada mensagem Hello corresponde a um pedido ou a uma resposta a um pedido. A estrutura das primitivas será descrita mais à frente.
- ROUTE_REQUEST – esta primitiva tem como objetivo a descoberta de nós específicos nas redondezas. Da forma como está arquitetada a solução, esta primitiva acaba por ser desnecessária, pois com o envio periódico das mensagens Hello todos os peers acabam por se conhecer entre si ao fim de um período de tempo.
- ROUTE_REPLY – esta primitiva é a resposta à primitiva anterior. Quando a primitiva ROUTE_REQUEST chega a um nó que conhece uma rota para o nó pretendido é devolvida um ROUTE_REPLY com essa informação.
- MESSAGE – esta primitiva serve para "transportar" mensagens entre os peers. É reencaminhada de nó em nó até chegar ao destinatário da mensagem.

2.2 Formato das mensagens protocolares (PDU)

Para além dos campos do cabeçalho que estão inerentes aos datagramas, particular destaque para o endereço de origem da mensagem que é utilizado pela aplicação, foi definida ainda uma estrutura fixa para cada primitiva com alguns campos específicos às funções que estas desempenham no âmbito da aplicação.
Existem algumas semelhanças entre os vários PDU's, mas existe algo que todos têm em comum, o primeiro campo reservado que identifica o tipo de primitiva que está a ser enviado e serve de base para o parsing da mensagem.

Em seguida serão especificados os PDU's de cada uma das primitivas:

HELLO

HELLO|MACHINENAME|TIPO|peerlist|

- MACHINENAME – identifica o nome do nó de origem da mensagem.
- TIPO – define o tipo de mensagem Hello que está a ser enviada e pode assumir dois valores: "Request" ou "Reply".
- Peerlist – lista de nós vizinhos que um determinado nó conhece. Este campo é apenas preenchido quando se trata de uma resposta a um pedido e tem a seguinte forma: "peername&leapsTABpeername&leaps..." em que peername é o nome do nó vizinho e leaps corresponde ao número de saltos que a rota tem até o atingir.

ROUTE_REQUEST

ROUTE_REQUEST|MACHINENAME|DESTINATION|PEER2FIND|JUMPS|PEERS_PATHOFRETURN

- MACHINENAME – identifica o nome do nó de origem da mensagem.
- DESTINATION – identifica o nome do nó de destino da mensagem.
- PEER2FIND – identifica o nome do nó para o qual se pretende encontrar um caminho.
- JUMPS – define o TTL da mensagem.
- PEERS_PATHOFRETURN – contém uma lista que vai sendo preenchida à medida que a mensagem vai sendo propagada pela rede. Esta lista contém apenas os nomes dos nós por onde a mensagem passou separados por tabs e tem como objetivo definir o caminho que uma possível mensagem de resposta terá de percorrer até chegar ao nó que fez a pesquisa original.

ROUTE_REPLY

ROUTE_REPLY|MACHINE|DESTINATION|PEER2FIND|ANSWER|LEAPS|PEERS_PATHOFRETURN

- MACHINENAME – identifica o nome do nó de origem da mensagem.
- DESTINATION – identifica o nome do nó de destino da mensagem.
- PEER2FIND – identifica o nome do nó para o qual se pretende encontrar um caminho
- ANSWER – código que identifica se o nó foi ou não encontrado. Pode tomar dois valores: "OK" e "NF" (Not Found). Neste momento este campo não está a ter grande utilidade uma vez que apenas uma reposta de pesquisa com sucesso ou o timeout pela falta desta despoletam acções nos nós que fizeram ROUTE_REQUEST's. Inicialmente a ideia é que se uma mensagem de insucesso fosse devolvida por cada um dos nós da lista de vizinhos de um nó, também este enviasse uma mensagem ao nó que lhe remeteu o ROUTE_REQUEST do insucesso na procura do nó. Posteriormente adoptou-se uma solução mais *stateless* em que a maior parte da informação circula nas mensagens e não está armazenada nos nós.
- LEAPS – número de saltos que uma mensagem enviada pelo nó que remeteu o ROUTE_REQUEST terá que fazer até chegar ao nó para o qual foi solicitado uma rota.
- PEERS_PATHOFRETURN – tem a mesma estrutura do campo homólogo na primitiva ROUTE_REQUEST. Enquanto no ROUTE_REQUEST cada vez que a mensagem é propagada num peer, o nome deste é adicionado a esta lista, nesta primitiva passa-se o contrário, cada vez que passa num peer o nome deste é removido da lista. Este campo tem o comportamento duma stack, num sentido são feitos "PUSHES" no sentido contrário são feitos "POPS".

MESSAGE

MESSAGE|SENDER|DESTINATION|TEXT

- SENDER - identifica o nome do remetente da mensagem. Esta campo é mantido inalterado na propagação da mensagem.

- DESTINATION - identifica o nome do destinatário da mensagem. Tal como o campo SENDER é mantido inalterado na propagação da mensagem.
- TEXT – texto da mensagem enviada pelo utilizador.

2.3 Interações

A aplicação desenvolvida pode ser subdividida em duas componentes que atuam ou simulam duas zonas distintas da pilha protocolar:

Construção de rede

Nesta subcomponente da aplicação, transparente ao utilizador, estão envolvidos os métodos de criação e gestão de rede.
Quando a aplicação é iniciada é enviada uma primitiva HELLO por broadcast para a porta 9999 de todas as máquinas que se consiga chegar com um TTL de 1. Essas máquinas, caso tenham uma instância da aplicação a correr, irão adicionar o nó que enviou o pedido à sua lista de vizinhos conhecidos e irão enviar uma resposta com uma lista dos vizinhos que conhecem que não são atingidos a partir do nó que enviou o request.
Este procedimento é bastante importante porque permite evitar listas de vizinhos redundantes, em que um nó Z tem na sua lista que atinge o nó X através do nó Y e o nó Y tem na sua lista que atinge o nó X através de Z.
Este procedimento repete-se de forma periódica de forma a garantir que, por um lado novos nós que apareçam entretanto sejam reconhecidos e adicionados à rede, e por outro lado, nós que sejam desconectados ou saiam do raio da rede sejam removidos das listas de vizinhos dos nós da rede.

Envio de mensagens

Quando a aplicação é iniciada é também colocado um processo "à escuta" por pedidos TCP na porta 9999 para que o utilizador possa enviar mensagens.
Quando uma mensagem é recebida por um nó, este começa por verificar se tem na sua lista de vizinhos o destinatário da mensagem. Caso não tenha é despoletado um ROUTE_REQUEST para tentar encontrar um caminho até este.
O ROUTE_REQUEST é enviado a todos os vizinhos conhecidos e, caso o nó vizinho desconheça também o nó pretendido, é depois propagado aos vizinhos dos vizinhos até que o nó seja encontrado, o TTL ou um timeout seja atingido. Se o nó for encontrado este é adicionado à lista de nós e o procedimento de envio de mensagem pode prosseguir.
É enviada uma primitiva MESSAGE ao destinatário ou então ao vizinho que sabe o caminho para o destinatário e esta será propagada de nó em nó até que chegue ao destinatário. De notar que no centro da rede as mensagens circulam todas sobre o protocolo UDP pelo que não existe garantia que as mensagens cheguem ao destino.
Ao chegar ao nó destino é depois enviada a mensagem a todos os "clientes" que tenham estabelecida uma conexão TCP à porta 9999 do nó.

3 Implementação

3.1 Detalhes

Na implementação da aplicação optou-se por separar cada uma das funções básicas da aplicação no sua própria classe, sendo que todas elas implementam a interface Runnable para poderem ser executadas na sua própria thread. As classes e respetivas funções são:

- NeighbourFind – esta classe permite despoletar RouteRequests para descobrir determinado peer

- NeighbourDiscoverer – serve para descobrir os peers na rede. Tem ainda como função remover os peers obsoletos da lista de peers conhecidos.
- MessageSockets – esta classe abre um socket para "escutar" pedidos TCP numa determinada porta. Quando uma mensagem é recebida obtém as informações necessárias da rota a utilizar e encaminha a mensagem por essa mesma rota
- NetworkListener – esta classe é responsável por processar todas as mensagens UDP que chegam à aplicação.

Para além destas classes existem ainda as classes associadas aos diferente tipos de primitivas que o sistema usa, nomeadamente: Hello, RouteReply, RouteRequest e Twitter.
Cada uma destas classes tem os métodos set e get associados às propriedades descritas anteriormente no formato das mensagens e ainda métodos para construir a string que é depois convertida em bytes e integrada nos datagramas e vice-versa, construir uma instância com base numa string.

Para além destas classes mais associadas à comunicação entre os vários nós, existem ainda uma outra classe chamada "Peers" bastante relevante ao contexto da aplicação. Esta classe tem os métodos necessários para gerir o repositório de peers dum nó permitindo adicionar nós e também remover nós obsoletos. Permite ainda alguma optimização de rotas, construindo uma lista de nós cuja rota tenha o mínimo de saltos possíveis.

3.2 Optimizações e problemas encontrados

Como já foi referido uma das optimizações que se introduziu foi a minimização do número de saltos de uma rota. Para que esse cálculo seja realizado, cada nó na lista de nós conhecidos tem a informação do número de saltos que são necessários dar até ser atingido para além do próximo nó dessa rota.
Quando, na resposta a uma mensagem Hello, são recebidos novos nós para adicionar à lista, caso o nó já seja conhecido apenas atualiza a informação do nó se o número de saltos for inferior ao já conhecido, caso contrário não atualiza a lista.

Uma situação que surgiu e também foi necessário resolver foi a redundância entre as listas de nós e os caminhos falsos que isso por vezes introduzia. Exemplo: o nó X chega ao nó Y através do nó Z, se o nó Y deixar de estar disponível este, pelos mecanismos já descritos, irá ser removido da lista de nós do nó Z. O que é necessário impedir é que o nó Z não seja depois atualizado com uma lista de nós do nó X e insira uma rota para Z por X.
Para resolver este problema foi resolvido por duas medidas: quando um nó vizinho é considerado obsoleto, todos os nós da lista cuja rota é por esse nó são também removidos. Por outro lado, na resposta à mensagem Hello, apenas são inseridos os nós cuja rota não inclua o nó que enviou o Hello em primeiro lugar.

Para evitar que os RouteRequests pudessem entrar em ciclo introduziu-se também uma proteção. Como já foi dito, à medida que o RouteRequest vai sendo propagado uma lista de peers vai sendo construída. Se um RouteRequest chegar a um nó que já consta dessa lista, esse nó vai simplesmente ignorar esse pedido. Esta otimização permite evitar o envio de algumas mensagens desnecessárias e assim poupar algum tráfego na rede.

3.3 Parâmetros

Todas as variáveis que definem o funcionamento da aplicação estão definidas na classe "Global", contudo neste momento estas não são configuráveis. São apenas propriedades estáticas com valores default configurados. Essas variáveis são:

- APP_PORT – define a porta de funcionamento da aplicação. Por defeito utiliza a porta 9999.
- MCAST_ADDR – define o endereço para o qual os broadcasts são enviados. Por defeito utiliza o endereço "FF02::1".

- HELLO_INTERVAL – define o intervalo em segundos entre dois broadcasts de mensagens HELLO. Por defeito utiliza o valor 120.
- DEAD_INTERVAL – define o intervalo de tempo em segundos desde o último contacto de um peer a partir do qual este é considerado obsoleto. O valor por defeito é 360.
- LEAP_COUNT – define o TTL utilizado nos RouteRequests.

De notar que os valores por defeito do HELLO_INTERVAL e do DEAD_INTERVAL foram definidos desta forma para se poder verificar o funcionamento dos RouteRequests. Em termos práticos o valor destes dois parâmetros seria muito mais baixo para que, apesar da maior carga que isso introduziria na rede, o sistema fosse mais reativo e a detecção de quebras de ligação fosse mais célere.
Uma vez que o cerne do sistema assenta sobre o protocolo UDP é necessário que a detecção de quebras de ligação seja o mais rápida possível para prevenir perdas de mensagens.

4 Testes e Resultados

Como todas as funções da aplicação são distribuídas por diferentes threads foi ainda possível criar uma mini aplicação na consola, com dois comandos apenas, find e peers que permitem obter informação acerca do estado dos peers em tempo útil.
Este dois comandos tornaram-se bastante úteis pois permitiram identificar de forma mais fácil e rápida problemas que surgiram durante o desenvolvimento.

Para a realização de testes foi utilizada a ferramenta core para simular uma rede, uma vez que seria impraticável reunir o número de máquinas necessárias para realizar os testes. A topologia utilizada foi a apresentada na imagem abaixo:

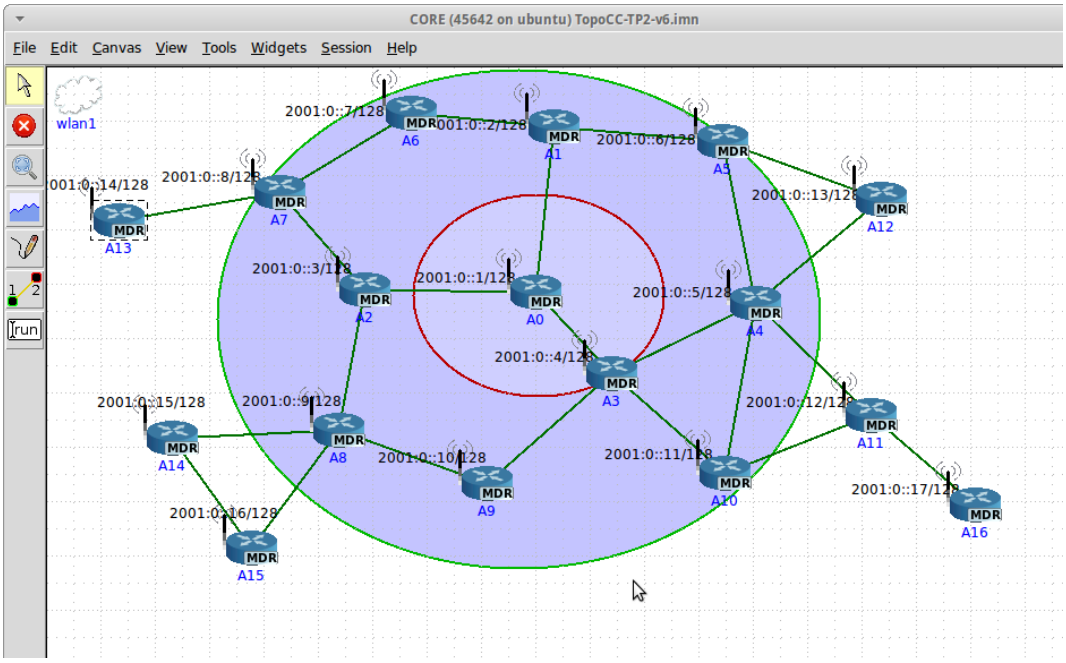

Figura 1 - Topologia da rede de testes no Core

O resultado de uma interação é apresentada na imagem abaixo. Na imagem abaixo a consola no canto superior esquerdo corresponde ao nó A13, a do canto superior direito corresponde ao nó A0, a do canto inferior direito corresponde ao nó A2 e do canto inferior esquerdo ao nó A7. As duas consolas no centro correspondem a duas conexões Telnet, a de cima é uma conexão ao nó A0 e a debaixo ao nó A13.

O que se pretendia testar neste caso de estudo era se a aplicação era capaz de enviar uma mensagem do nó A0 para o A13, sendo que após o arranque do sistema o nó A0 não tem conhecimento de nenhuma rota para a A13 como se pode comprovar pela execução do comando peers na consola associada ao A0.
Após se iniciar a aplicação adhoc em todos os terminais, abriram-se as conexões telnet aos respetivos nós e enviou-se para o nó A0 a seguinte mensagem "TW #A13 Mensagem linda". Para que o nó A0 consiga enviar a mensagem ao nó A13 tem que começar por descobrir a rota. Para isso envia um RouteRequest ao único nó que conhece, A2. O nó A2, como não tem conhecimento de nenhuma rota para o A13, propaga o RouteRequest para os nós que conhece, A0 e A7. O nó A0 como já está incluído na lista de peers do RouteRequest simplesmente ignora a mensagem, contudo o nó A7, que tem uma rota para A13, responde com um RouteReply para o A2 que depois será também reencaminhado para A0. Desta forma o nó A0 tem agora uma rota por A2 para o nó A13 e pode por isso enviar a mensagem.
Como se pode ver na consola com a sessão telnet ao nó A13 a mensagem foi recebida com sucesso.

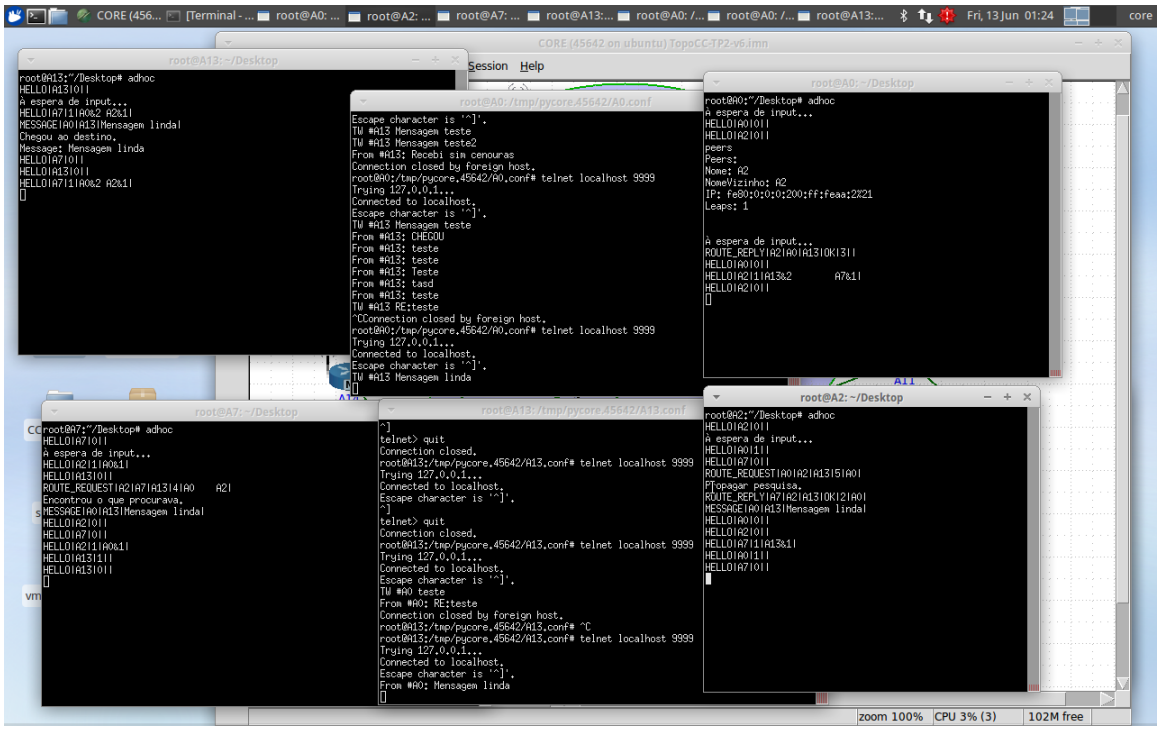

Figura 2 - Exemplo de uma interação com a aplicação

5 Conclusões

Após a realização deste trabalho pode-se verificar que o JAVA dispõe de inúmeras bibliotecas que permitem de forma relativamente rápida desenvolver aplicações que comunicam entre si, apesar de estarem em diferentes máquinas.
Para além disso pode-se comprovar quão complexos e quantos pormenores existem no cálculo de rotas em redes. Desde problemas de ciclos que se podem estabelecer a detecção de falhas passando pela detecção de novos nós e pela remoção de nós obsoletos, há aqui uma série de funcionalidades nas redes que são de tal forma transparentes que raramente nos apercebemos delas, mas que não deixam de ser vitais ao funcionamento das redes.
Finalmente, pode-se afirmar que os objetivos do trabalho foram cumpridos. Desenvolveu-se uma aplicação capaz de encontrar outros peers em funcionamento na rede e enviar mensagens entre eles. Isto sobre uma rede adhoc de formação espontânea e ainda com a capacidade de lidar com alguns dos problemas descritos anteriormente.

6 Trabalho futuro

Ao nível do funcionamento da aplicação, poder-se ia introduzir uma cache negativa de route requests com um determinado timeout. As operações de RouteRequest são bastante pesadas e envolvem muitas comunicações entre vários peers, especialmente no caso de insucesso, em que o critério de paragem é apenas o TTL. Para evitar alguma desta carga poder-se-ia criar uma cache negativa para que se dois RouteRequests fossem feitos de seguida para um mesmo peer não contactável, no segundo pedido não houvesse propagação do pedido.

Por outro lado, a aplicação poderia ser também capaz de guardar rotas alternativas para que, no caso de uma das rotas ficar indisponível, uma rota secundaria pudesse ser utilizada para enviar mensagens.

Ao nível da aplicação em si, poder-se-ia dotar a aplicação da capacidade de enviar ficheiros, lidando com os problemas que isso introduz (segmentação de pacotes e gestão de pacotes) e ainda a capacidade de definir os parâmetros de configuração da aplicação através do arranque da aplicação com parâmetros ou através da leitura de um ficheiro de configurações.

Poder-se-ia ainda permitir que através do envio de mensagens através do Telnet se conseguisse obter informações relativas ao estado do nó, basicamente corresponderia a passar a invocação dos métodos find e peers da consola para o Telnet.