An Asynchronous Leader Election Algorithm for Dynamic Networks without Perfect Clocks

Daniel Almeida
Email: up201306450@fe.up.pt

Eliseu Pereira
Email: up201303855@fe.up.pt

Miguel Bertão
Email: up201305771@fe.up.pt

***Resumo*—Neste relatório pretende-se apresentar a implementação de um algoritmo de eleição numa rede assíncrona e dinâmica, baseado no artigo "*An Asynchronous Leader Election Algorithm for Dynamic Networks*". O algoritmo desenvolvido apresenta algumas diferenças à proposta do artigo. Contudo, o objetivo do mesmo passa por assegurar que, independentemente das mudanças de topologia que possam ocorrer numa rede de processos interligados, todos os nós apresentem um líder único. O procedimento de eleição de líder está assente na constante monitorização da arquitetura da rede e na comunicação realizada via UDP *Multicast* caso alguma variação ocorra.**

I. Descrição do problema

Em sistemas distribuídos, um algoritmo de eleição de líder tem como função designar um único processo numa rede de vários computadores como o principal decisor e organizador de tarefas. Após a execução de um destes algoritmos, numa rede de múltiplos nós interligados, deve existir apenas um líder e todos os outros processos devem ter conhecimento de qual o processo líder.

A. Aplicações reais

O dinamismo de uma rede pode gerar alguns obstáculos ao funcionamento da mesma, isto porque é necessário detetar as frequentes formações e falhas de ligações e atuar de acordo com esses eventos. Redes móveis são um exemplo de redes que sofrem constantes mudanças na sua topologia, especialmente se a rede for *wireless*, onde há maior probabilidade de interferência e/ou falha nas ligações. Como este, existem muitos outros exemplos de redes dinâmicas, surgindo assim a necessidade de desenvolver algoritmos que consigam lidar com este género de problemas.

Um outro problema associado às interações em sistemas distribuídos assenta na identificação das relações causais e cronológicas entre as várias ações. De forma a evitar estes problemas e a diminuir a dependência do acesso a um relógio global, adota-se muitas vezes o uso dos relógios lógicos. Estes fazem parte de uma classe de algoritmos que conseguem assegurar a sincronização dos eventos e a consistência interna dos relógios. No entanto, esta implementação não foi realizada neste projeto.

B. TORA

A solução implementada é baseada no algoritmo presente em [1], que tem como base o *Temporally Ordered Routing Algorithm* (TORA). Apesar da forte adoção de conceitos e rotinas descritos nesse artigo, foram realizadas algumas alterações com objetivo de facilitar a sua implementação.

O *Temporally Ordered Routing Algorithm*, ao contrário do usual neste tipo de algoritmos, não recorre ao uso do caminho mais curto para solucionar o controlo de propagação das mensagens. Este cria e usa, na raiz do nó de destino, uma DAG (Directed Acyclic Graph), fazendo com que a informação a ser transmitida só flua de nós com alturas maiores para nós com alturas menores. Desta maneira, a informação flui apenas de montante para jusante, impedindo a ocorrência de *loops* nas comunicações, assim como a disseminação desnecessária de informação, já que a informação não pode ser transmitida para nós com alturas superiores.

C. Funcionamento Global

Neste algoritmo, o líder é escolhido após uma pesquisa completa pela rede. A pesquisa consiste na troca de múltiplas mensagens entre os nós via UDP *Multicast*, em que cada um destes nós envia informação sobre si e sobre o seu estado atual para todos os seus vizinhos. Este conjunto de dados enviados de vizinho para vizinho relativos a cada nó é denominado por altura do nó. Aquando da receção de uma mensagem de um vizinho, o nó poderá ter de alterar a sua altura. Por exemplo, quando ocorre uma mudança de líder, o novo líder terá de propagar pela rede o seu valor identificativo de forma a que todos os outros nós tenham conhecimento desta importante alteração, atualizando um dos parâmetros da sua altura. Contudo, esta troca de mensagens é causada pela alteração da topologia inicial da rede. Isto é, se houver uma quebra de ligação por parte do líder atual é absolutamente necessário eleger um novo coordenador, e assim se inicia um novo processo de eleição. Para situações opostas, onde surgem novos nós e ligações, é também necessário que o novo nó da rede crie a sua própria altura, considerando todos os aspetos da rede em que está agora inserido.

II. Alturas (heights)

A principal base deste algoritmo é um vetor constituído por diferentes valores que são intitulados de alturas. Como se pode observar na tabela I este vetor é constituído por 7 valores, sendo 2 deles *timestamps*. Estas *timestamps* são baseadas no relógio local do computador onde está a decorrer o processo. Cada um destes parâmetros serão utilizados para realizar as mais diversas verificações e comparações ao longo do processo de pesquisa e eleição que é descrito no capítulo IV.

Tabela I
CONSTITUIÇÃO DO VETOR DE ALTURAS

Byte número	0	1	2	3	4	5	6
Alturas	s	oid	r	d	nlts	lid	id

- **s:** 0 ou *timestamp* positiva do início da pesquisa
- **oid:** 0 ou ID do nó que começou a pesquisa
- **r:** 0 caso ainda não tenha atingido o *deadend* 1 caso contrário
- **d:** distância atual para o líder
- **nlts:** 0 ou *timestamp* do momento da eleição do líder
- **lid:** ID do atual líder
- **id:** ID do próprio nó

Ao longo deste relatório vão ser referidas inúmeras vezes tanto o par de líder como os níveis de referência, sendo que o par de líder é constituído pelos valores do líder ID (lid) e do *timestamp* de eleição deste mesmo (nlts). Os níveis de referência são constituídos pelos 3 primeiros valores do vetor, sendo estes o *timestamp* referente ao início da pesquisa (s), o ID do nó que começou a pesquisa (oid) e o valor atual do *deadend*.

III. COMUNICAÇÕES

No que diz respeito às comunicações, estas foram implementadas de modo a serem dinâmicas, ou seja, com uma rede já criada e com um líder eleito, ser possível ir acrescentado nós a essa mesma rede. A comunicação utiliza *multicast*, pois para redes com um elevado número de conexões é mais eficiente disseminar a informação desta maneira. Quanto à camada de transporte, esta utiliza UDP pois implica menores *overheads*, apesar da menor fiabilidade comparado com TCP.

A. Sockets

Como foi referido acima, neste projeto foram utilizados *sockets multicast*. Estes utilizam endereços na gama de 224.0.0.0 até 239.255.255.255. Cada nó da rede tem um endereço atribuído da seguinte forma: os primeiros 3 valores são os mesmos para todos os nós e o último valor é constituído pelo ID do próprio nó. Por exemplo, para o nó de ID 5 seria atribuído o endereço "225.0.0.5". Assim, este nó vai criar um *socket* de receção e de envio para este endereço. O *socket* de receção apenas é utilizado para adicionar novas conexões com outros nós (mensagens de conexão), enquanto que o *socket* de envio serve tanto para enviar mensagens de *update* para nós que tenham uma conexão com este nó, como para enviar mensagens de *alive*. Basicamente, para um novo nó se ligar a outro já pertencente a uma rede, apenas tem de indicar o ID do nó ao qual se quer conectar durante a fase da sua configuração.

B. Protocolo

Anteriormente, foram feitas inúmeras referências a diferentes tipos de mensagens. Nesta secção vão ser explicadas essas mesmas mensagens e também o protocolo utilizado.

A tabela II apresenta os 3 tipos de mensagens, bem como a constituição de cada uma. Quanto à mensagem de *connect* ela é transmitida apenas quando um nó se pretende ligar a outro, contendo o ID do nó ao qual se quer ligar e o valor da altura do nó que envia a mensagem. No que diz respeito às mensagens de *alive*, estas têm como objetivo fazer *reset* ao *timeout* implementado ao nível do *socket*, de forma a este não ser ativado sem ter ocorrido nenhuma falha. Finalmente, as mensagens de *update* são responsáveis por transmitir a informação (alturas) entre os nós quando está a ocorrer uma pesquisa.

Tabela II
TIPOS DE MENSAGENS UTILIZADAS

Tipo de Mensagem	No para conectar-se	Vetor de alturas
connect	ID do nó a conectar	Alturas do nó
alive	0	0
update	0	Alturas do nó

O protocolo que verifica estas mensagens está implementado dentro de um ciclo que percorre o mapa de conexões dos vizinhos, verificando se existe alguma mensagem. O *socket* de receção do próprio nó também está presente neste mapa pois é necessário verificar se existem nós a quererem iniciar uma conexão.

C. Link Up

Como podemos observar na figura 1, a transição entre uma conexão estar UP ou DOWN é feita utilizando as funções de *Link Up* e *Link Down*. No que diz respeito à função *Link Up*, esta é executada quando é recebido um pedido de conexão. Assim, durante a sua execução é criado um novo *socket* para receber informação (*updates*) da nova ligação. Para além disso, é executado o algoritmo base para verificar o líder ou a necessidade de iniciar uma pesquisa.

D. Link Down

Quanto à função *Link Down* esta é executada quando é ativado o *timeout* do respetivo *socket*. De seguida é removida a conexão do mapa e é verificado se existe algum vizinho, caso não exista nenhum vizinho este nó elege-se líder, caso contrário verifica as condições necessárias para iniciar uma pesquisa - se não se verificarem, continua a sua execução funcionamento.

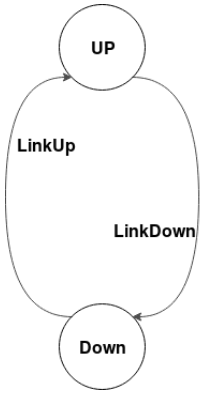

Figura 1. Transições entre UP e DOWN

E. Deteção de falhas

A deteção de falhas neste projeto recorre a *timeouts*. Assim, quando passa um certo intervalo de tempo entre mensagens seja de *alive*, seja de *update*, o *timeout* é ativado. Assim sempre que um nó falha, o vizinho que deteta esse *timeout* executa a função *Link Down*.

IV. Algoritmo Implementado

Inicialmente, quando surge uma nova conexão, isto é, quando um processo j é iniciado e se conecta ao processo i, uma mensagem do tipo *connect* é enviada por j para i. Este último nó, ao detetar a nova ligação, dissemina para o seu grupo *multicast* uma mensagem *update* com a altura de j. A partir deste momento, sempre que ocorre qualquer alteração na altura de um nó, são novamente enviadas mensagens deste género. A perda de conexão com o líder pode ser outro fator que obriga à atualização da altura dos seus vizinhos. Cada *update* desencadeia uma sequência de comparações entre a altura do processo recetor da mensagem e o conteúdo da mesma. De acordo com o resultado dessas verificações, o processo em questão pode ou não adotar certos parâmetros da altura de j. Por fim, ao nível *core* do algoritmo, passa também a ser necessário verificar a altura de todos os vizinhos do processo, de modo a este tomar uma decisão final. A máquina de estados relativa ao algoritmo executado em cada processo pode ser observada na figura 2.

Analisando a máquina de estados, é de notar que um nó i se localiza no estado inicial ("INIT"), caso não se encontre a conferir um *update*.

Quando finalmente é recebida uma mensagem *update* do processo j, é realizada a comparação, entre i e j, dos dois valores referentes ao líder atual, que correspondem à marca temporal que indica quando o líder foi eleito e o respetivo id. Uma diferença entre este par de valores da altura dos dois processos significa que, recentemente, ocorreu uma eleição de um novo líder e é necessário o processo i adotar esse líder, alterando esses parâmetros na sua altura ("AdoptLP"). No fim desta alteração, o processo volta ao estado inicial. Caso os valores sejam iguais, prossegue-se com o algoritmo.

Neste ponto do algoritmo é realizada a verificação "SINK", que consiste num conjunto de sub-verificações. A completa confirmação desta condição implica que uma pesquisa está em curso e o nó i terá de realizar o seu papel nessa mesma pesquisa. Em primeiro lugar, é necessário confirmar que i e todos os seus vizinhos possuem o mesmo par de líder. Caso exista alguma diferença, é possível que já tenha ocorrido uma eleição que o nó i ainda não tenha conhecimento e desta forma, o algoritmo não poderá continuar. É também mandatório que o nó i não seja o líder, visto que nesta condição não se poderia iniciar uma pesquisa. Por fim, verifica-se se a altura de i é menor que a altura de qualquer um dos seus vizinhos. Cada parâmetro presente na tabela I é verificado individualmente, existindo uma ordem crescente de prioridade da esquerda para a direita. O facto de i não ser líder e apresentar a menor altura entre os seus vizinhos é um fator indicativo da necessidade de i propagar a sua altura de modo aos vizinhos atualizarem a sua. Isto acontece porque o nó líder é o nó com menor altura em toda a rede, que apesar dos três primeiros parâmetros serem iguais em todos os nós, apresenta a menor distância ao líder (4º parâmetro da altura), que é 0. Ou seja, todos os nós não líderes, em situação normal, apresentam pelo menos um vizinho com altura menor que a sua. Para estes casos o "SINK" não será verificado e i volta ao seu estado inicial.

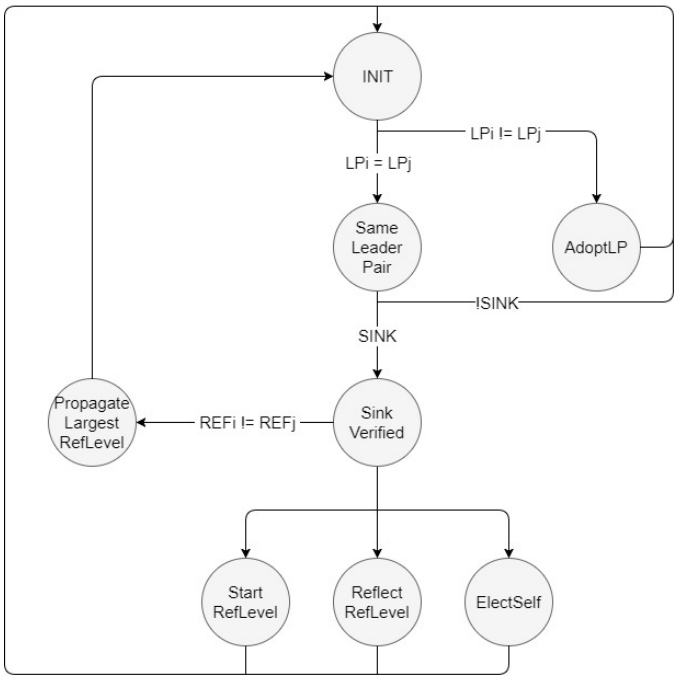

Figura 2. Máquina de estados

Após todas estas condições confirmadas, é seguro afirmar que está a decorrer um processo de pesquisa e eleição de líder no sistema. Assim, o nó i terá de analisar em concreto os valores da altura de j contidos na mensagem *update* recebida, com o objetivo de entender em que momento da pesquisa ele se encontra. Esta análise mais pormenorizada vai ser o pilar da decisão final tomada pelo nó i que pode ser um dos seguintes casos:

- **PropagateLargestRefLevel**: este estado apenas é ativado caso os valores de referência da altura de i e j sejam diferentes. Os parâmetros de referência são a constante temporal que marca início da pesquisa, o ID do nó que iniciou esse mesmo processo e o valor de *deadend*, que indica se a rede foi totalmente percorrida (1) ou não (0). O facto destes três valores serem diferentes entre os dois vizinhos indica que i ainda não tem conhecimento da pesquisa e irá agora atualizar esses três parâmetros na sua altura. Ainda neste estado, i realiza uma procura entre os seus vizinhos para descobrir a distância ao nó que iniciou a pesquisa, tendo i de atualizar esse parâmetro na sua altura da seguinte forma: $di = min\{dj \ \forall \ j \in Vi\} - 1$. Esta operação é crucial, já que quando se chega ao fim da rede, o sentido de envio de mensagens é invertido e isto permite que o "SINK" continue a ser verificado.

Contudo, caso os valores de referência sejam iguais um dos próximos estados é que será ativado.

- **ReflectRefLevel**: este estado durante a pesquisa apenas é ativado caso o nó em questão seja o último da rede. Ou seja, apercebe-se que é o nó mais distante do iniciador da pesquisa e que o *deadend* ainda se encontra a 0, alterando o valor deste para 1 e atualizando o seu quarto parâmetro $di = 0$, de modo a inverter o sentido da pesquisa, ficando com esse parâmetro maior que os seus vizinhos (que o têm negativo graças ao "PropagateLargestRefLevel").
- **ElectSelf**: aqui é observado que o *deadend* está 1 e que o nó i iniciou a pesquisa ($OIDi = IDi$). Isto significa que a rede foi totalmente pesquisada e é o momento de i se eleger como líder da rede. Para isto, os quatro primeiros parâmetros são repostos a 0 e o $LIDi = IDi$. A atualização final da altura de i é também propagada para toda a rede.
- **StartRefLevel**: a nível de código apenas se chega a este estado caso todas as outras condições até aqui descritas não tenham sido verificadas. Apenas ocorre uma vez durante o processo de eleição e é o estado onde é iniciado esse mesmo processo, registando a marca temporal de início e atualizando a altura conforme necessário ($OIDi = IDi$ e $di = 0$).

Após finalizado o processo de qualquer um destes estados, o nó i volta ao seu estado inicial. Por fim, é de salientar que este processo algorítmico é realizado diversas vezes para cada nó numa só pesquisa.

V. Resultados

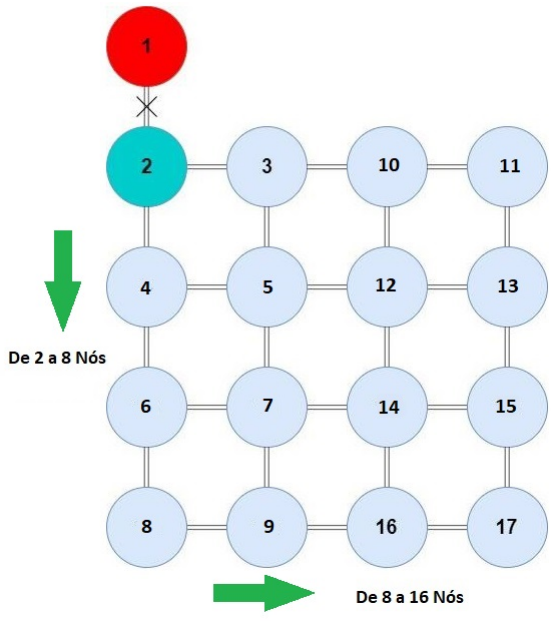

Figura 3. Rede utilizada para testes

Um algoritmo de eleição tem como mais importante propriedade o tempo que demora a eleger um novo líder, caso não exista um na rede. Essa duração é definida por dois fatores: o número de ligações existentes entre o nó que iniciou a pesquisa e o último nó da rede e pelo número de ligações totais nessa rede, já que quanto maior esse valor, mais rápida será a disseminação de informação. Foi pensada numa topologia de rede em que esses dois fatores estivessem fortemente presentes, que pode ser observada na figura 3.

Tabela III
Duração de cada diferente topologia de teste

Número de nós	Número de ligações	Duração (ms)
2	1	2
4	4	2
6	7	3
8	10	7
10	13	8
16	24	10

A rede completa (16 nós) foi apenas um dos testes, já que foram também realizados testes para 2, 4, 6, 8 e 10 nós. Na tabela III é possível observar o número de ligações por cada teste e reparar que, mesmo que a rede seja maior, um grande aumento do número de ligações faz com que a duração de uma eleição não varie da mesma forma.

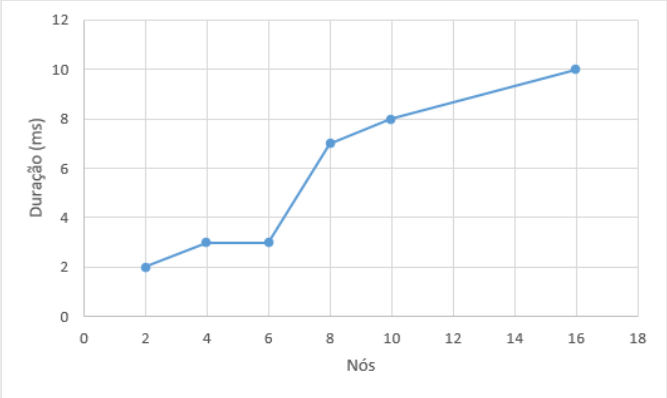

Figura 4. Duração de eleição nas diferentes topologias de rede

VI. Análise Critica

Neste projeto, do ponto de vista de performance, adotamos como linguagem de programação o C++, pois em geral é mais rápido na execução de instruções devido a ser uma linguagem mais próxima da linguagem máquina. Esta execução mais rápida permite-nos obter melhores resultados ao nível de tempo de eleição. Apesar disso, com a utilização de C++ perdeu-se mais tempo na implementação, pois para programar uma certa funcionalidade requer um maior número de instruções do que Java.

Do ponto de vista das comunicações, seria mais eficiente utilizar apenas um endereço *multicast* em toda a rede, apesar de neste algoritmo não ser possível, pois cada nó possui vizinhos, que são os únicos nós da rede que necessitam da sua informação. Ainda nas comunicações é de salientar que o *multicast* é mais eficiente neste tipo de algoritmo, pois é necessário partilhar a informação com vários nós ao mesmo

tempo. Quanto à utilização de UDP, justifica-se devido aos baixos *overheads* deste protocolo. Além de existir alguma redundância nas mensagens que são enviadas, por exemplo, se se perder uma mensagem de *alive*, o nó não é considerado como "morto" automaticamente, pois é necessário passarem 10 segundos, que correspondem à perda de 10 mensagens de *alive*, para essa consideração se confirmar.

A nível de algoritmo de pesquisa e eleição é de salientar a sua simples implementação após a sua total compreensão. Contudo, essa compreensão já não é tão fácil de atingir devido ao elevado número de especificações e verificações necessárias para que ocorra uma eleição com sucesso. Este algoritmo implementado está já bastante otimizado, visto que todas as mensagens do tipo *update* são essenciais para a correta comunicação entre nós. No entanto, há sempre melhorias a fazer, por exemplo, implementar a possibilidade de cada nó ter um completo conhecimento da rede, isto é, das alturas de todos os nós, sejam eles vizinhos ou não. Para grandes redes isto seria impraticável já que ocuparia grandes porções de memória e o número de mensagens trocadas seria enorme, mas para redes com um baixo número de nós e ligações seria uma propriedade que poderia ser útil dependendo da aplicação. Por fim, é importante referir novamente que este algoritmo apresenta melhores resultados para redes com elevado número de conexões do que para redes longitudinais com baixo número de ligações, o que pode ser um fator bastante positivo para redes *mesh*.

VII. Conclusão

Em suma, é possível afirmar que a implementação do algoritmo proposto foi realizado com sucesso e o objetivo principal de ter um sistema de eleição de líder numa rede assíncrona e dinâmica em funcionamento foi atingido.

Caso seja necessário verificar o código implementado, este encontra-se hospedado na página do *github* na seguinte hiperligação [2].

A nível de código, o trabalho foi dividido em dois, comunicação e algoritmo de pesquisa e eleição. Este último esteve ao encargo de Eliseu Pereira, enquanto a comunicação e integração da mesma no problema foi realizada por Daniel Almeida e Miguel Bertão. Contudo, quando se procedeu à junção destas duas tarefas ainda havia bastante trabalho pela a frente e a partir desse momento o projeto foi desenvolvido de igual forma pelos três membros do grupo. Posto isto, diria que a percentagem atribuída a cada membro deve ser de 33,33%.

Referências

[1] R. Ingram, P. Shields, J. E. Walter, and J. L. Welch, "An asynchronous leader election algorithm for dynamic networks," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[2] Repositório Github, https://github.com/eliseu10/LeaderElection/.

[3] A. Derhab and N. Badache, "A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 7, pp. 926–939, 2008.