

Introdução:

O problema de eleição de coordenador consiste em dado um conjunto de processos, estabelecer entre eles um coordenador, de forma que só haja um coordenador. Este coordenador vai ter um papel diferenciado dos demais processos. Sempre deve haver um coordenador, então caso um coordenador caia outro processo deve assumir. Para que outro processo venha assumir deve haver uma eleição. O processo eleito deve ser único e todos os processos devem concordar entre si quem é o eleito. O processo eleito será, dentre os processos ativos, o que tiver maior identificador.

Os critérios para uma eleição funcionar corretamente são de segurança, um processo P participante de eleição pode não conhecer o eleito, ou conhecer um eleito ao final da execução, também é importante a subsistência, todos os processos participam da eleição e determinam o eleito, ou falham.

No algoritmo do valentão parte-se da ideia de que as mensagens são todas confiáveis, assim são permitidas falhas na eleição, o sistema deve ser síncrono devido aos tempos utilizados para detecção de falhas. Aqui todos os processos se conhecem, ao contrário do algoritmo de anel que só se conhece a vizinhança. São utilizadas 3 mensagens básicas no Valentão, eleição, resposta e coordenador. Estas mensagens, respectivamente, são responsáveis por perguntar se um processo de maior identificador está ativo, responder à um processo de menor identificador que ele não é o eleito, o processo eleito manda essa mensagem para os outros processos. A seguir um exemplo de eleição retirado de [1]:

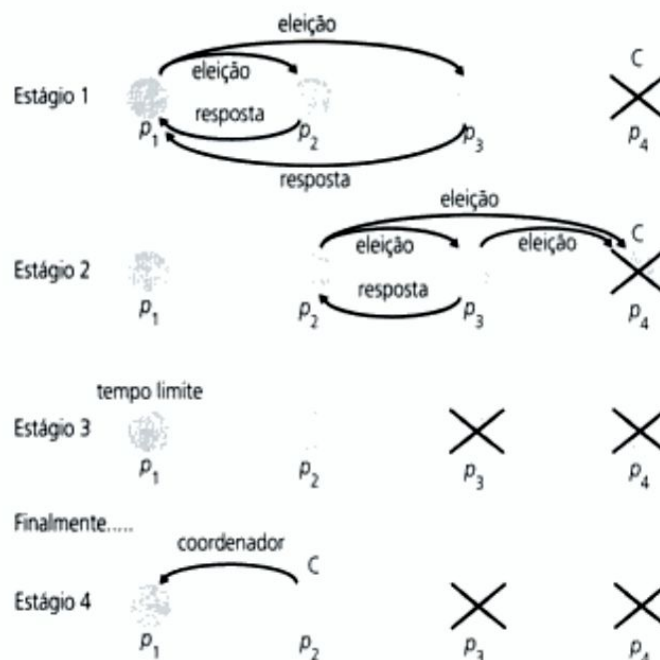


Figura 12.8 O algoritmo valentão.

Nesse caso P_3 falha antes de se declarar eleito, sendo assim após o estouro do tempo limite P_2 declara que é coordenador para o processo com o identificador menor do que o dele. Note que na falha de P_3 , P_2 ao notar e por saber que é o processo ativo com maior identificador, já envia a mensagem de coordenador à P_1 .

Implementação:

Na implementação foram enfrentados vários desafios devido a falta de experiência do aluno em programação com sockets. O primeiro desafio foi em pensar o que fazer para que independente da topologia todos os nós da rede se conhecessem sem ter que dar como entrada o IP de cada um. Foi decidido criar um Rastreador (Classe Tracker), onde cada nó se comunicava com um servidor e este dava a lista de quem estava conectado com ele, assim foi resolvido o problema do conhecimento da topologia.

Assim, cada cliente iniciaria o processo (Classe Process). O processo deveria enviar e responder mensagens então era necessário um server socket para cada um, e para gerenciar as conexões com o processo foi criado uma thread de gerência (Classe ProcessManager) que basicamente ficava em laço aceitando novas conexões, e para cada conexão lançava uma thread para tratar as mensagens (Classe ProcessController). A partir de então a comunicação podia ser feita, os processos então enviavam keep alive (Método send) para o coordenador. Quando um processo enviava um keep alive havia uma espera, caso a espera estourasse e não houvesse um ACK, iniciava-se o processo de eleição (Método election), se o ACK fosse recebido o thread que detectava a mensagem notificava quebrando a espera e o processo enviava outro keep alive (O ciclo continuava). O processo que começasse uma eleição iria então enviar uma mensagem, “?” equivalente à election, para todos os processos com identificador maior. Quando ativos, os processos maiores que recebiam “?”, respondiam aos menores “N” indicando que os menores não poderiam se eleger. Caso algum processo maior tivesse se declarado coordenador, mas outro processo havia iniciado a eleição, ao receber “?” o novo coordenador envia uma mensagem “b” indicando que ele é o novo coordenador. Se um processo que iniciou eleição não receber resposta dos outros com maior identificador que ele, este envia para os processos de menor identificador “b”. Quem iniciou eleição e foi avisado que não pode ser eleito fica esperando um novo coordenador mandar uma mensagem, caso o tempo passe, este processo irá realizar outra eleição.

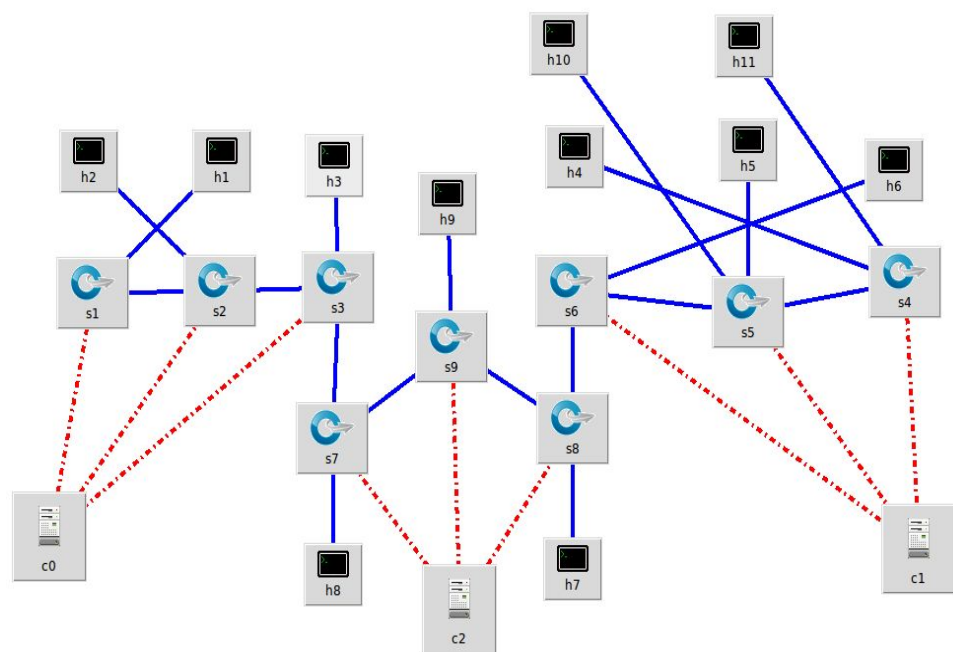
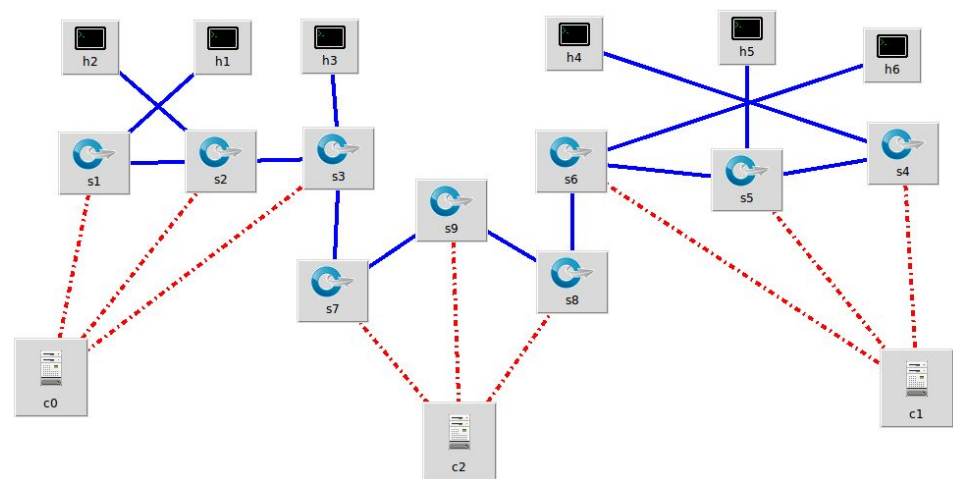
Assim, para casos em que os processos que caem não voltam o algoritmo já convergia. O problema era quando os processos voltavam. Para esse caso foi necessário cuidar dos seguintes casos:

1. Coordenador de maior identificador volta;
2. Ex-Coordenador de identificador menor que o Coordenador atual volta;
3. Cliente volta achando que o coordenador é outro;

Para o primeiro caso e terceiro caso bastou que o processo que estava voltando iniciasse uma eleição, assim ele avisaria a todos que ele era o coordenador ou conheceria o novo coordenador. Para o segundo caso, foi necessário desativar a flag de eleito (booleano isBoss) antes de derrubar o processo em questão, assim quando ele voltasse iria iniciar uma eleição como nos demais casos. Foi então resolvido o problema de queda e volta.

Testes e Resultados:

Os testes foram realizados nas duas topologias a seguir, onde a primeira é uma versão simplificada e sem atraso da segunda. Em nenhuma das topologias há perda de pacote.



Essas imagens foram retiradas do miniedit, também é importante falar que o delay que existe na segunda topologia estão nos enlaces (s1,s2), (s2,s3), (s3,s7), (s8,s6), (s6,s5), (s5,s4), onde para cada enlace citado há um atraso de 10 milissegundos.

No mininet foi gerado um script em python para cada topologia, assim era possível utilizar aquela topologia através do comando “sudo python topologia.py”. No próprio miniedit se pode editar o atraso de cada enlace e a taxa de perda de pacotes, assim como outros parâmetros. Sendo assim a única coisa que fica faltando no script é iniciar em cada host a aplicação. A aplicação em um host é iniciada através da instrução ‘host.sendCmd([Comando])’.

À seguir como exemplo segue a parte do meu script responsável por isso:

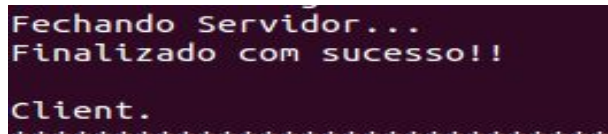
```
info( '*** Post configure switches and hosts\n')
h1.sendCmd('java -jar tracker.jar')
time.sleep(3)
h2.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h3.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h4.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h5.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h6.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h7.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h8.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h9.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h10.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
h11.sendCmd('java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1')
print h1.waitOutput()
print h2.waitOutput()
print h3.waitOutput()
print h4.waitOutput()
print h5.waitOutput()
print h6.waitOutput()
print h7.waitOutput()
print h8.waitOutput()
print h9.waitOutput()
print h11.waitOutput()
print h10.waitOutput()
```

Esse foi o exemplo apresentado em sala onde eu utilizo 10 nós da rede para aplicar o bully. A cada 5 segundos é sorteada a chance de 50% do coordenador cair, e não há chances dele voltar, os clientes não tem chance de cair também.

As instruções “print host.waitOutput()” servem para que o impressão da execução só seja escrita na tela após toda execução e em ordem.

O tempo de espera ociosa (sleep) que é colocado no script, é para que o servidor rastreador inicialize antes dos clientes para que não haja problemas na execução.

Lembrando que a topologia deve estar no mesmo diretório que os jars para que tudo funcione. Também é importante avisar que ao executar o script depois de alguns segundos irá aparecer essa mensagem indicando que o servidor Tracker foi finalizado, não é nenhum erro, é só esperar 45 segundos que os resultados serão impressos.



```
Fechando Servidor...
Finalizado com sucesso!!

Client.
*****
```

Na pasta contida este trabalho seguem 4 scripts que podem ser rodados se na maquina estiver instalado o mininet. Os dois primeiros seguem a primeira topologia, os dois ultimos a segunda topologia.

A diferença entre cada um é:

- topo1.py :
 - 'java -jar hostCustomizavel.jar 5 5 75 95 100 10.0.0.1';
 - Significa que são 5 nós utilizados na rede;
 - A cada 5 segundos há sorteio de queda;
 - Há 25% de chance de um coordenador cair;
 - Há 5% de chance de um nó que caiu, voltar;
 - Não há chances de um cliente cair;
- topo2.py :
 - 'java -jar hostCustomizavel.jar 5 5 75 1 50 10.0.0.1';
 - Significa que são 5 nós utilizados na rede;
 - A cada 5 segundos há sorteio de queda;
 - Há 25% de chance de um coordenador cair;
 - Há 99% de chance de um nó que caiu, voltar;
 - Há 50% de chance de um cliente cair;
- topo3.py :
 - 'java -jar hostCustomizavel.jar 10 5 50 80 90 10.0.0.1';
 - Significa que são 10 nós utilizados na rede;
 - A cada 5 segundos há sorteio de queda;
 - Há 50% de chance de um coordenador cair;
 - Há 20% de chance de um nó que caiu, voltar;
 - Há 10% de chance de um cliente cair;
- topo4.py :
 - 'java -jar hostCustomizavel.jar 10 5 50 100 100 10.0.0.1';
 - Significa que são 10 nós utilizados na rede;
 - A cada 5 segundos há sorteio de queda;
 - Há 50% de chance de um coordenador cair;
 - Não há chance de um nó que caiu, voltar;
 - Não há chance de um cliente cair;

Conclusão:

Acerca do trabalho, para todos os testes realizados sempre houve consistência de quem era o coordenador, porem não houve uma tentativa com perca de pacotes já que no

trabalho não foi explorada a possibilidade de envio de múltiplas mensagens. O algoritmo, em teoria, deve funcionar em uma topologia com perca de pacotes, só que deve haver um tempo de convergência bem maior. Para otimizar seria bom implementar a possibilidade de enviar múltiplas mensagens antes de decidir começar uma eleição.

Foi uma ótima oportunidade para aprender a trabalhar com sockets de uma maneira mais aprofundada, ao fazer essa aplicação distribuída tive uma maré de problemas que tiveram que ser solucionados gerando conhecimento. Creio ter tido uma ótima experiência que irá me auxiliar em trabalhos futuros.