Implementação do Algoritmo Bully Election

Anthony Emanoel de Albuquerque Jatobá[1], **Júlio César Ferreira Silva de Holanda** [2]

[1]Instituto de Computação – Universidade Federal de Alagoas (UFAL)

aeaj@ic.ufal.br, jcfsh@ic.ufal.br

Abstract. *The aim of this report is to describe an implementation of the Bully Election leader election algorithm made with the Castalia simulator for the Wireless Sensor Networks (WSNs) discipline of the Computer Science course taught by Professor André Aquino.*

Resumo. *Este relatório tem por objetivo descrever uma implementação do algoritmo de eleição de líder Bully Election feita com o simulador Castalia, para a disciplina de Redes de Sensores Sem Fios (RSSFs), do curso de Ciência da Computação, ministrada pelo professor André Aquino.*

1. Eleição de Líder

Diversas aplicações no contexto de Sistemas Distribuídos requerem certo grau de coordenação entre os processos da rede. Em uma rede completamente descentralizada, os nós podem divergir acerca de decisões, precisando se comunicar para chegar a um consenso. Esse consenso é muitas vezes difícil e custoso de se atingir. Nesses casos, podemos centralizar esse poder de escolha a um nó da rede.

Eleição de líder é o processo de designar um nó para desempenhar um papel especial na aplicação, como o de coordenar as tarefas dos nós na rede [Balhara and Khanna 2014]. Algoritmos de eleição de líder descrevem o processo para a escolha desse nó líder, de forma que não haja dois nós líderes na rede ao mesmo tempo e que todos os nós ativos na rede o conheçam.

1.1. Bully Election

O algoritmo de Bully Election [Garcia-Molina 1982] foi uma das primeiras soluções para o problema da eleição de líder. Nesta seção vamos descrever o funcionamento básico do algoritmo. Cada nó na rede possui um ID numérico associado e está conectado a todos os demais. Este ID é o critério de prioridade do algoritmo, isto é, o nó de maior ID dentre os participantes da eleição será eleito.

A eleição tem início quando um nó percebe a falha no líder. Após perceber esta falha, o nó contata todos os outros nós da rede com maior prioridade que a sua. Se algum desses nós responde, o nó retira sua candidatura e aguarda a mensagem informando o líder. Caso contrário, o nó assume que todos os nós de prioridade superior à sua estão falhos e, portanto, deve ser eleito líder. Então, o nó envia uma mensagem informando aos demais que ele é o novo líder.

2. Implementação

A implementação do algoritmo se deu exclusivamente na camada de aplicação. A seguir resumimos o trabalho realizado em cada arquivo. O manual [Boulis 2009] foi particularmente útil nesta tarefa. O código fonte se encontra disponível em <https://github.com/anthonyjatoba/castalia-bully>.

2.1. BullyElection.h e BullyElection.cc

No arquivo de cabeçalho da aplicação, definimos os *timers*. Estes *timers* definem eventos que são disparados em função do tempo:

- `SEND_HEARTBEAT`: envio periódico da mensagem *heartbeat*, sinal enviado pelo líder indicando seu correto funcionamento;
- `CHECK_LEADER`: verificação de funcionamento do líder. O nó verifica se o último *heartbeat* se deu dentro do intervalo de tempo esperado;
- `CHECK_ELECTION`: verificação do resultado da eleição;
- `FAILURE`: na ocorrência deste evento, o nó tem uma probabilidade de falhar ou, caso se encontre falho, se recuperar.

Além disso, foram definidas as variáveis privadas (usadas pelos nós para coordenar as tarefas de eleição):

- `lastHeartbeat`: timestamp do último heartbeat recebido;
- `leaderID`: ID do líder;
- `applying`: indica se o nó está participando de uma eleição;
- `oks`: número de *OKs* recebidos;

Alguns métodos implementados e suas responsabilidades são listadas a seguir. Os métodos de criação e envio de pacotes foram omitidos.

- `startup()`: define os *timers* e o primeiro líder propaga seu id para a rede;
- `fromNetworkLayer(...)`: ações tomadas na chegada de pacotes;
- `callElection()`: nó entra em estado de eleição e envia o pacote para os nós de ID superior ao seu;
- `failureUtility()`: evento que calcula se o nó entrou em estado de falha ou recuperou-se da falha;
- `timerFiredCallback(...)`: dispara os eventos associados a *timers*.

2.2. BullyElectionMessage.msg

No arquivo de descrição dos pacotes, adicionamos um campo extra com um *enum* indicando o tipo da mensagem. Listamos a seguir as possibilidades:

- `HEARTBEAT`: o líder está funcionando a contento;
- `ELECTION`: o nó está se candidatando a líder;
- `OKAY`: o nó recebeu a mensagem de eleição e vai assumir a partir daí;
- `LEADER`: o nó avisa aos demais que foi eleito líder.

2.3. VirtualApplication.h e VirtualApplication.cc

Algumas variáveis foram adicionadas às classes base de aplicação, para que pudéssemos defini-las na configuração do experimento. São elas:

- `isLeader`: indica se o nó é o líder atual.
- `numNodes`: cada nó deve saber o número de nós na rede. Dessa forma, pode enviar as mensagens de eleição aos nós de prioridade superior.
- `working`: variável que indica se o nó está funcionando.
- `failureP` e `recoveryP`: probabilidade de um nó falhar e de um nó falho se recuperar, respectivamente.

3. Resultados e Discussões

Para avaliar o algoritmo, definimos dois cenários de configurações: no primeiro cenário, **variamos o número de nós** da rede; no segundo cenário, variamos tanto a **probabilidade de falha** quanto a **probabilidade de recuperação** dos nós.

Os nós foram dispostos de maneira aleatória em uma área de 10x10 metros. O nosso intuito foi manter todos os nós ao alcance uns dos outros para testar o algoritmo sem nos preocupar com aspectos de roteamento.

Cada experimento teve duração de 1800 segundos e foi executado 10 vezes, para os resultados não serem sujeitos ao acaso. O cálculo de média e intervalo de confiança foi feito pelo utilitário **CastaliaResults**, distribuído junto ao Castalia. Os gráficos foram gerados com o utilitário **CastaliaPlot**.

3.1. Variação do número de nós

Nesta configuração, executamos o experimento variando o número de nós da rede, que pode ser 4, 8, 12, 16, 20 e 24. As probabilidades de falha e recuperação foram fixadas em 5% e 12%, respectivamente.

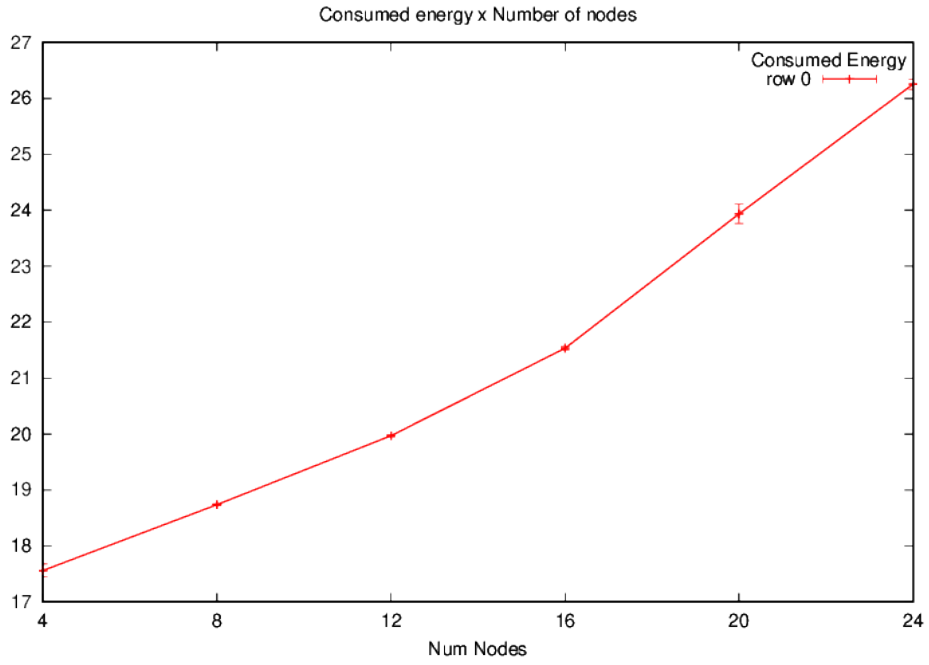

Figura 1. Energia consumida

O gráfico da *figura 1* ilustra o consumo de energia conforme variamos o número de nós. É possível notar um crescimento no consumo, pois conforme a rede cresce, cada nó deve se comunicar com um número maior de nó na ocorrência de uma eleição. Podemos notar também uma leve curvatura no consumo observado, o que condiz com a intuição.

Na *figura 2* temos os pacotes enviados em cada cenário. É possível notar um comportamento similar ao consumo de energia na quantidade e proporção de cada tipo de pacote, com exceção do tipo de pacote SYNC que se mantém praticamente estável em

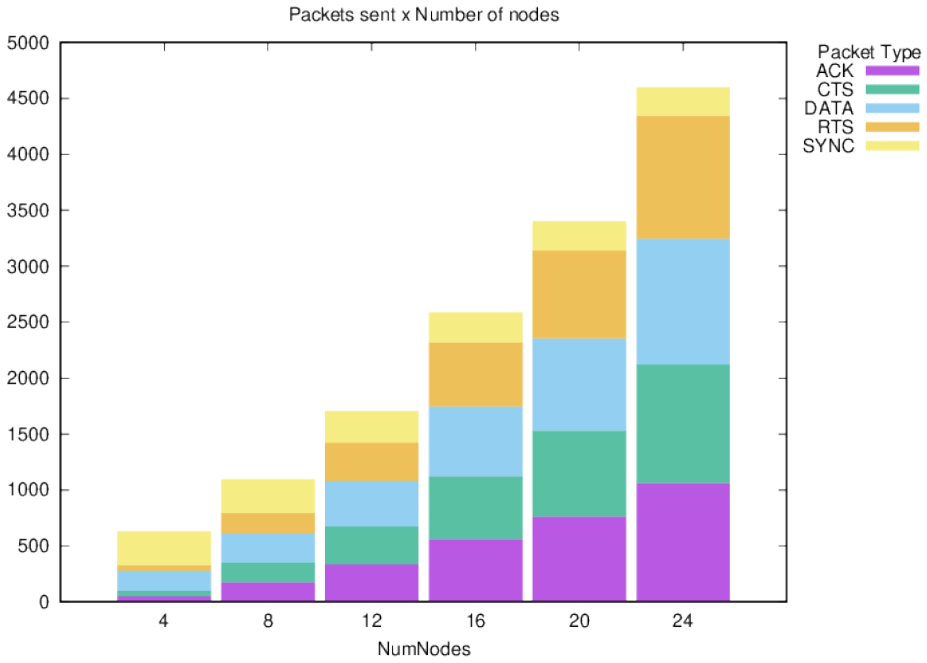

Figura 2. Pacotes enviados

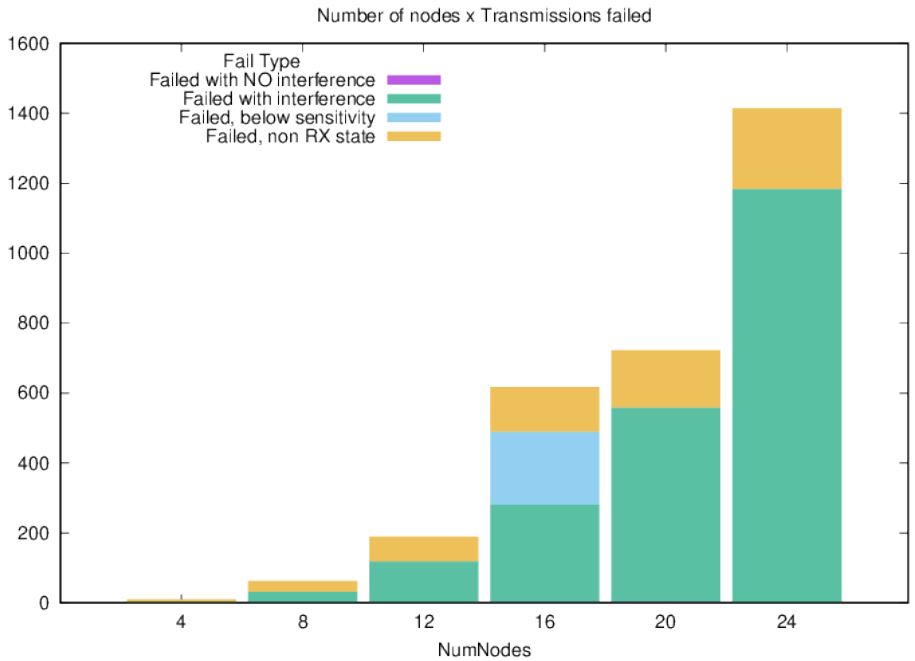

Figura 3. Falhas identificadas

cada configuração. Este tipo de pacote é usado para a sincronização dos nós no início dos cenários.

O gráfico da *figura 3* apresenta um comportamento interessante. Aumentando o número de nós, observamos mais interferência, o que leva a ocorrência de mais falhas. Isso torna-se bastante visível no cenário de 24 nós.

3.2. Variação das probabilidades de falha e recuperação

Neste experimento, variamos as probabilidades de falha e recuperação de cada nó do sistema. Os valores de falha testados são 2%, 5% e 10%, enquanto que os de recuperação são 4%, 8% e 12%. Todas as combinações destes valores são testadas.

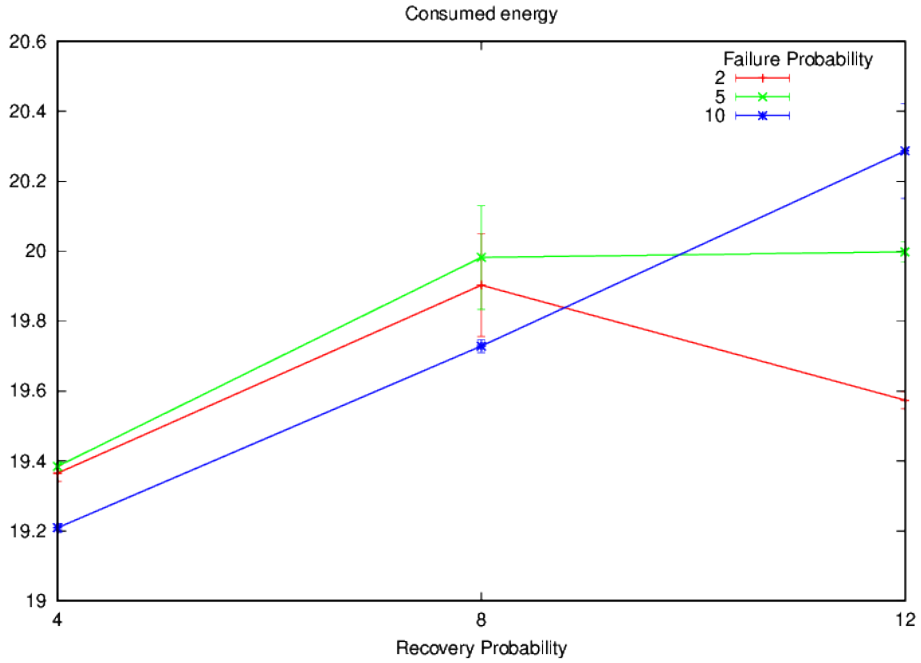

Figura 4. Energia consumida

É possível notar, através da *figura 4*, algumas tendências. No cenário com maior probabilidade de falha (linha azul, 10%), conforme aumentamos a probabilidade de recuperação, mais energia é gasta. Este é o cenário em que os nós estão mais frequentemente falhando e retornando de falhas, disparando mais eleições.

Nos cenários com falhas menos comuns, vemos um consumo maior de energia nos cenários iniciais, pois os nós estão ativos numa maior parte do tempo. Vemos esta tendência crescer e em seguida, se estabilizar ou decair.

A *figura 5* mostra os pacotes enviados nesta configuração. Nos cenários com baixa probabilidade de recuperação (4%), o aumento da chance de falha diminui o número total de pacotes enviados, pois os nós se encontram falhos com maior frequência, dificilmente retornando ao funcionamento.

Nos cenários com maior chance de falha (8 e 10%), aumentar a chance de recuperação faz com que mais pacotes sejam trocados. Isto se dá porque os pacotes estão mais frequentemente retornando de estados de falha e disparando eleições.

4. Conclusão

O trabalho presente permitiu que aplicássemos os conhecimentos adquiridos durante a disciplina de Redes de Sensores Sem Fio. Por meio desta prática, muitos conceitos e ideias - inclusive de outras disciplinas, como Sistemas Distribuídos e Redes de Computadores - foram melhor entendidos e internalizados, consistindo num complemento importante para a nossa formação.

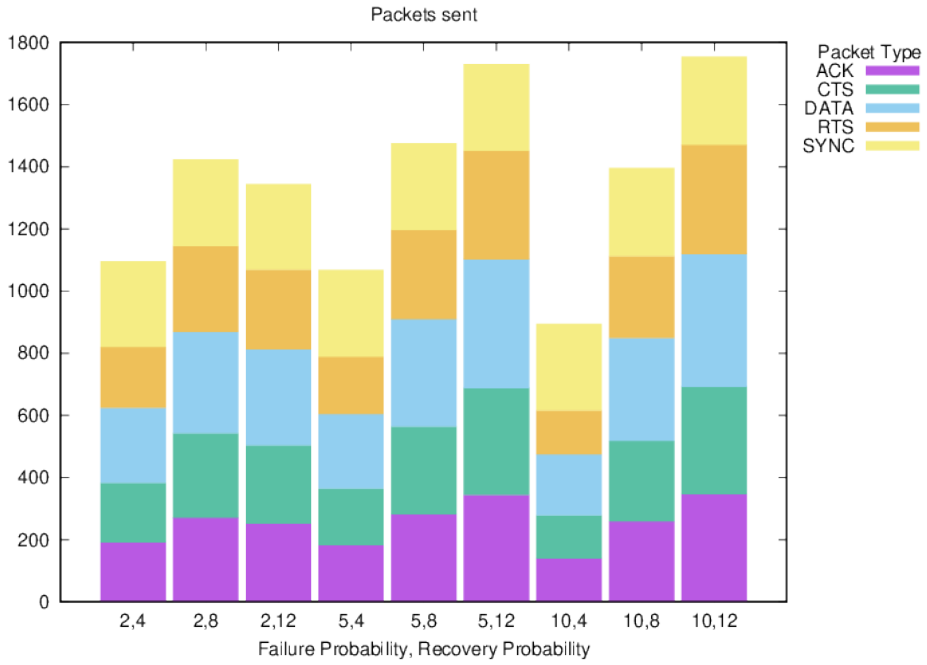

Figura 5. Pacotes enviados

Referências

Balhara, S. and Khanna, K. (2014). Leader election algorithms in distributed systems. *Journal of Computer Science and Information Technology, IJCSMC*, 3(6):374–379.

Boulis, A. (2009). Castalia user manual. *Online: http://castalia. npc. nicta. com. au/pdfs/Castalia-User Manual. pdf*.

Garcia-Molina, H. (1982). Elections in a distributed computing system. *IEEE transactions on Computers*, (1):48–59.