Algoritmos para Eleição de Líder em Sistemas Distribuídos

Jucele França de Alencar

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos para Eleição de Líder em Sistemas Distribuídos

Jucele França de Alencar [1]

Abril de 1998

Banca Examinadora:

- Prof. Dr. Ricardo de Oliveira Anido (Orientador)
- Prof. Dr. Valmir C. Barbosa
  Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ
- Prof. Dr. Cid Carvalho de Souza
  Instituto de Computação - UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação - UNICAMP

[1] **Apoio financeiro CNPq e FAPESP.**

BIBLIOTECA CENTRAL

UNIDADE BC
N.º CHAMADA:
V. Ex.
TOMBO BC/34199
PROC. 395/98
C D X
PREÇO R$ 11,00
DATA 10/06/98
N.º CPD

CM-00112604-9

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Alencar, Jucele França

AL25a Algoritmos para eleição de líder em sistemas distribuídos / Jucele França Alencar -- Campinas, [S.P. :s.n.], 1998.

Orientador : Ricardo de Oliveira Anido

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas operacionais distribuídos (Computadores). 2. Algoritmos. 3. Processamento distribuído. I. Anido, Ricardo de Oliveira. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Algoritmos para Eleição de Líder em Sistemas Distribuídos

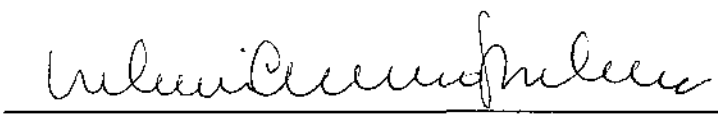
Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Jucele França de Alencar e aprovada pela Banca Examinadora.

Campinas, 6 de Abril de 1998.

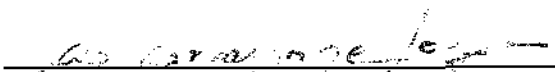
Prof. Dr. Ricardo de Oliveira Anido
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 17 de fevereiro de 1998 pela Banca Examinadora composta pelos Professores Doutores



Prof. Dr. Valmir Carneiro Barbosa



Prof. Dr. Cid Carvalho de Souza

Prof. Dr. Ricardo de Oliviera Anido

Agradecimentos

A Deus por tudo.

A meu orientador Prof. Ricardo Anido pela forma como me auxiliou no desenvolvimento deste trabalho.

A meu marido Francisco por seu companheirismo, apoio, paciência e carinho.

A meus pais e irmãos pelo incentivo e amor que sempre me deram apesar da distância.

A minha prima Senilde, seu marido Ricardo e seus filhos que me fizeram sentir em casa e me proporcionaram o tão valioso convívio familiar.

A todos os colegas pelo ambiente de harmonia, pela troca de experiências e pelos momentos de descontração.

MUITO OBRIGADA.

Resumo

A principal característica de sistemas distribuídos é permitir que vários computadores, situados em diferentes localizações, trabalhem em conjunto para a realização de uma tarefa, possibilitando o compartilhamento de recursos e a distribuição dos serviços. No entanto, devido à dificuldade de se obter uma consistência global do sistema, o desenvolvimento de algoritmos distribuídos torna-se uma tarefa bastante complexa. Visando facilitar o controle, algoritmos distribuídos necessitam, muitas vezes, de um processador que exerça uma função diferenciada dos demais numa dada aplicação. Este processador é normalmente chamado de coordenador ou líder, e o problema de eleição de líder consiste em criar um algoritmo distribuído para uma dada rede de processadores tal que, ao terminar a execução deste algoritmo um, e apenas um, dos processadores seja designado *líder*.

Esta dissertação apresenta um *survey* englobando as principais soluções para o problema de eleição de líder, analisando suas complexidades de mensagens e tempo e buscando uma classificação de modo a agrupar os algoritmos baseando-se no modelo do sistema e no princípio básico usado no desenvolvimento dos mesmos.

Abstract

The main feature of a distributed system is to allow many computers, at different locations, to work together performing a common task, making the sharing of resources and the distribution of services possible. However, due to the difficulty of acquiring a global view of the system, the development of distributed algorithms is very complex. Several distributed applications make use of a distinguished processor to execute some special task. This processor generally is called coordinator or leader; the leader election problem consists of finding a distributed algorithm for a given network such that, at the end of the execution, only one processor has been declared leader.

This dissertation presents a survey of the main algorithms for the leader election problem, analyzing their message and time complexities and attempting to classify the algorithms according to the system model and the basic principle used in their development.

Conteúdo

Agradecimentos v

Resumo vi

Abstract vii

1 Introdução 1
- 1.1 Conceitos Básicos . . . 1
  - 1.1.1 Algoritmos Distribuídos . . . 2
  - 1.1.2 Algoritmos para Eleição de Líder . . . 4
  - 1.1.3 Relação com outros tipos de problemas . . . 4
- 1.2 Estrutura da dissertação . . . 5

2 Algoritmos Clássicos 6
- 2.1 Algoritmo de LeLann (1977) . . . 6
- 2.2 Algoritmo de Chang e Roberts (1979) . . . 9
- 2.3 Algoritmo de Peterson (1982) . . . 12
- 2.4 Algoritmo de Garcia-Molina (1982) . . . 15
- 2.5 Alguns Limites Inferiores . . . 16

3 Uso de Senso de Direção 19
- 3.1 Eleição em Rede Completa . . . 20
  - 3.1.1 Algoritmo de Loui et al. . . . 20
  - 3.1.2 Algoritmo de Singh . . . 23
- 3.2 Eleição em Hipercubos . . . 29
  - 3.2.1 Algoritmo de Flocchini e Mans . . . 30
  - 3.2.2 Algoritmo de Tel . . . 35

4 Algoritmos Auto-estabilizantes 39
- 4.1 Introdução . . . 39
- 4.2 Eleição em Anel Uniforme . . . 40
  - 4.2.1 Algoritmo de Huang . . . 40

4.2.2 Algoritmo de Itkis et al. 44
4.3 Eleição em Redes Genéricas 56

5 Algoritmos Probabilísticos **61**
5.1 Algoritmos probabilísticos em redes anônimas 61
5.1.1 Com conhecimento do número de nós da rede 62
5.1.2 Sem conhecimento sobre o tamanho da rede 68
5.2 Algoritmos probabilísticos em redes não anônimas 75

6 Conclusão **80**

Bibliografia **84**

Lista de Tabelas

2.1 Tabela com alguns limites inferiores para complexidade de mensagens para o problema de eleição de líder. 18

6.1 Tabela com tabela com todos os algoritmos apresentados e suas complexidades. . 83

Lista de Figuras

2.1 Algoritmo de LeLann para anel unidirecional. 7
2.2 Exemplo de funcionamento do algoritmo de LeLann. 8
2.3 Algoritmo de Chang e Roberts. 10
2.4 Um exemplo de funcionamento do algoritmo de Chang e Roberts. 11
2.5 Algoritmo de Peterson. 13
2.6 Um exemplo de funcionamento do algoritmo de Peterson. 14
2.7 Algoritmo de Garcia-Molina. 17
2.8 Um exemplo de funcionamento do algoritmo de Garcia-Molina. 18

3.1 Exemplo de uma rede completa com senso de direção. 19
3.2 Algoritmo de Loui et al. para redes completas com senso de direção. 22
3.3 Exemplo de funcionamento do algoritmo de Loui et al. para redes completas com senso de direção. 23
3.4 Exemplo de rede dividida em 4 sub-anéis. 24
3.5 Algoritmo de Singh para redes completas com senso de direção (parte 1). 26
3.6 Algoritmo de Singh para redes completas com senso de direção (parte 2). 27
3.7 Algoritmo de Singh para redes completas com senso de direção (parte 3). 28
3.8 Modelos de senso de direção para hipercubos: (a) dimensão (b) distância. 30
3.9 Exemplos de caminhos entre os processadores i e j (<1,2,0>, <2,0,1> e <0,1,0,2,0>). 31
3.10 Algoritmo de Flocchini e Mans para hipercubos com senso de direção baseado na dimensão. 33
3.11 Exemplo de funcionamento do algoritmo de Flocchini e Mans. 34
3.12 Algoritmo de Tel para hipercubos com senso de direção baseado na dimensão. . . 37
3.13 Exemplo de funcionamento do algoritmo de Tel. 38

4.1 Exemplo de funcionamento do algoritmo auto-estabilizante de Dijkstra. O processador destacado é o processador base e as setas indicam os processadores que podem utilizar o recurso. 41
4.2 Regras do algoritmo de Huang. 42
4.3 Outra forma de escrever as regras do algoritmo de Huang. 42
4.4 Exemplo de funcionamento do algoritmo de Huang. 43
4.5 Exemplo de anel subdividido em segmentos. 44

4.6 Algoritmo auto-estabilizante de Itkis et al. para anel uniforme. 46
4.7 Regra 1. 47
4.8 Regra 2A. 48
4.9 Regra 2B. 49
4.10 Regra 3. 50
4.11 Regra 4. 50
4.12 Regra 5. 51
4.13 Regra 6. 52
4.14 Regra 7. 53
4.15 Regra 8. 54
4.16 Regra 9. 54
4.17 Exemplo de funcionamento do algoritmo de Itkis et al. 55
4.18 Algoritmo auto-estabilizante para redes genéricas. 59
4.19 Exemplo de funcionamento do algoritmo auto-estabilizante para eleição em redes genéricas de Dolev et al. 60

5.1 Algoritmo síncrono para redes anônimas com n conhecido. 63
5.2 Exemplo de funcionamento do algoritmo síncrono de Itai e Rodeh com n conhecido. 64
5.3 Algoritmo assíncrono para redes anônimas com n conhecido. 66
5.4 Exemplo de uma iteração do algoritmo assíncrono de Itai e Rodeh com conhecimento de n. 67
5.5 Algoritmo síncrono para redes anônimas com $N \leq n < 2N$. 69
5.6 Exemplo de uma iteração do algoritmo síncrono de Itai e Rodeh com $N \leq n < 2N$. 70
5.7 Algoritmo de Matias e Afek para redes anônimas sem conhecimento de n. 72
5.8 Exemplo de funcionamento do algoritmo de Matias e Afek para redes anônimas genéricas. 73
5.9 Algoritmo de Itai para anel anônimo. 76
5.10 Algoritmo básico de Chang ou Vernadat et al. 77
5.11 Exemplo de funcionamento do algoritmo básico de Chang ou Vernadat et al. . . 78
5.12 Exemplo de funcionamento do algoritmo de Chang com alguns processadores possuindo maiores probabilidades de vencer. Os pseudo-processadores estão representados na mesma cor e próximos dos processadores reais a que estão associados. 78

Capítulo 1

Introdução

A principal característica de sistemas distribuídos é permitir que vários computadores, com localizações físicas diversas, trabalhem em conjunto para a realização de uma tarefa. Isso possibilita a distribuição dos serviços de uma dada aplicação, o compartilhamento de dados e recursos, a replicação de informação, aumentando a disponibilidade, além de facilitar a interação entre as pessoas que fazem uso desse sistema.

Entretanto, o desenvolvimento de *software* para ambientes distribuídos é muito mais complexo do que para ambientes centralizados, devido à dificuldade de se conseguir uma consistência global do sistema acarretada pela distribuição do controle e a autonomia dos componentes. Em um sistema distribuído não se pode estabelecer um estado global do sistema inteiro.

Muitas vezes, visando facilitar o controle, algoritmos desenvolvidos para ambientes distribuídos necessitam de um processador que exerça uma função diferenciada dos demais numa dada aplicação. Este processador é normalmente chamado de coordenador ou líder. Quando o coordenador corrente falha, o sistema, para continuar em operação, deve escolher um novo coordenador.

O problema de eleição de líder consiste em encontrar um algoritmo distribuído para uma dada rede de processadores tal que, ao terminar a execução do algoritmo um, e apenas um, dos processadores seja designado *líder*.

A motivação para este trabalho foi a de se realizar uma ampla pesquisa das diversas abordagens encontradas para a resolução do problema de eleição de líder em sistemas distribuídos. Esta dissertação trata-se então, de um *survey* englobando, além das soluções clássicas, os principais trabalhos publicados recentemente, analisando suas complexidades de mensagens e tempo e buscando uma classificação de modo a agrupar os algoritmos com base no modelo do sistema e no princípio básico usado no desenvolvimento dos mesmos.

1.1 Conceitos Básicos

Nesta seção são apresentados alguns conceitos relativos a sistemas distribuídos que serão utilizados ao se analisar os algoritmos no decorrer da dissertação.

1.1.1 Algoritmos Distribuídos

Sistemas distribuídos são formados por vários processadores (também chamados nós) interligados através de uma rede de comunicação. Um algoritmo distribuído é composto de processos e canais de comunicação entre estes processos. Como, normalmente, para os problemas que estamos estudando, existe um único processo executando em cada processador, em alguns casos usamos o termo processo indicando o processador em que este está em execução, como se cada processador estivesse associado a um único processo. Devemos no entanto salientar que, em sistemas distribuídos, há diversas aplicações executando ao mesmo tempo e, em cada nó, um conjunto de processos executam concorrentemente.

Os sistemas distribuídos possuem características que influenciam no desenvolvimento de aplicações para estes sistemas. Algumas dessas características são: a topologia da rede, o mecanismo de comunicação, a uniformidade do sistema, existência de identificadores para os processadores, uso de senso de direção e definição do tipo de falhas a serem tratadas.

Os canais de comunicação possuem uma estrutura caracterizando a conectividade da rede. Esta estrutura é chamada *topologia de comunicação da rede* e especifica como estão ligados os processadores. Em geral usamos um grafo para representar a rede de comunicação permitindo uma melhor visualização das ligações entre os processadores, onde os vértices simbolizam os processadores e as arestas, os canais de comunicação.

As topologias mais comuns são: anel, estrela, árvore, malha, hipercubo e completa. Embora redes completas muitas vezes pareçam impraticáveis, devido ao grande número de conexões, elas são muito estudadas para estabelecer limites de complexidades para o caso de redes mais práticas (com menos conectividade).

Todos os trabalhos apresentados nesta dissertação consideram que os canais de comunicação são livres de falhas. Isso significa que, em geral, as seguintes propriedades são obedecidas:

- Não há duplicação de mensagens.
- Não há corrupção de mensagens.
- Os canais de comunicação são *FIFO* (*First In First Out*), ou seja, as mensagens transmitidas entre dois processadores são recebidas obedecendo a ordem de envio das mesmas.
- O tempo para que uma mensagem enviada seja recebida pode ser indeterminado, porém deve ser finito.
- Se o tempo para que uma mensagem enviada seja recebida for limitado, pode-se afirmar que se após o tempo determinado a mensagem não for entregue a seu destino, esta deve ter se perdido e não será mais entregue.

Outra importante característica dos sistemas distribuídos é que eles podem ser *síncronos* ou *assíncronos* [IR90]. No sistema síncrono há um relógio global que controla a comunicação e as ações de todos os processadores, onde o tempo é divido em intervalos finitos. Os processadores iniciam sua execução no intervalo de tempo $t = 0$. A cada intervalo de tempo t um processador

p_i lê uma mensagem enviada no intervalo de tempo $t-1$ se essa mensagem existe, executa alguma tarefa e pode enviar no máximo uma mensagem a cada processador. Portanto, havendo sincronismo é possível dividir o algoritmo em ações que são executadas simultaneamente por todos os processadores. Já no caso assíncrono, cada processador possui seu próprio relógio sem relação com os relógios dos demais processadores, podendo executar a taxas não constantes; assim, os diferentes processadores podem ser uns mais lentos que outros. Não existindo um sincronismo não é determinado quanto tempo levará para uma mensagem enviada ser recebida. Assim, no caso assíncrono, não há relação entre as execuções dos diversos processadores.

Em uma rede os processadores podem ter identificações. Uma rede é denominada *anônima* quando seus processadores não possuem identificações que podem ser usadas em suas computações, nem mesmo para identificar o processador origem ou destino de uma mensagem. Como citado em [IR90] e demonstrado em [Ang80] não há algoritmo para eleição de líder em rede anônima que execute em todos os vértices e termine em tempo finito. O fato é que a simetria da rede não pode ser quebrada sem que haja computação infinita ou resultado incorreto.

Uma forma de quebrar a simetria de uma rede é possibilitar a distinção de seus canais de comunicação de acordo com algum esquema global definido. Essa diferenciação dos canais é definida como *senso de direção.*

Outro mecanismo usado para quebrar a simetria da rede é diferenciar os processadores. Um sistema é dito *uniforme* se todos seus processadores executam o mesmo programa. Caso exista um processador diferente e todos os outros idênticos, o sistema é conhecido como *semi-uniforme.*

A *terminação* de um algoritmo distribuído é definida como o estado em que todos os processos terminaram de executar o que deveriam e não há mais nenhuma mensagem em trânsito (mensagens enviadas e ainda não recebidas).

Um conceito interessante também usado é o de *auto-estabilização.* Um algoritmo auto-estabilizante é aquele que inicia sua execução em qualquer estado, possivelmente impróprio para os parâmetros estabelecidos para o problema, e ao executar, sem intervenção externa, se estabiliza em um estado desejado.

Para realizar a comparação entre diferentes algoritmos que solucionam um mesmo tipo de problema é preciso estabelecer uma forma de medir o consumo de recursos dos algoritmos. Esse consumo de recursos pode ser medido de várias maneiras:

- Complexidade de mensagens: corresponde ao número total de mensagens trocadas pelo algoritmo, ou seja o número máximo de mensagens enviadas no pior caso para todas as variações possíveis na execução do algoritmo. Esta medida pode ser substituída por uma mais exata a complexidade de mensagens em bits, que pode ser útil para mostrar diferenças relevantes entre os algoritmos quando o tamanho das mensagens depende do número de nós na rede ou do número de canais de comunicação entre os nós. A complexidade de mensagens em bits leva em conta o número de mensagens trocadas pelo algoritmo e seus tamanhos.

- Complexidade de tempo: é dada pelo tempo gasto pela computação do algoritmo, considerando que o tempo para processamento de um evento é zero e o tempo de transmissão

é de pelo menos uma unidade de tempo.

- Complexidade de espaço: é definida como sendo a quantidade de memória necessária para a execução do algoritmo.

1.1.2 Algoritmos para Eleição de Líder

Eleição de líder é um dos problemas mais estudados no campo de pesquisa de algoritmos distribuídos devido à sua importância prática. Assim são gerados diversos trabalhos e pesquisas para melhorar as complexidades dos algoritmos para os diferentes modelos computacionais.

Como já foi dito, o estabelecimento de um líder em um conjunto de processadores deve-se ao fato de, em certas aplicações, existirem tarefas especiais que devem ser cumpridas por um único processador. Algumas dessas funções especiais que o líder pode exercer são: realizar a exclusão mútua com controle centralizado, manter um grafo de espera global para detecção de *deadlock*, repor uma ficha perdida, controlar a entrada e saída de um dispositivo no sistema, ou servir como coordenador de uma tarefa.

Existem várias definições para o problema de eleição de líder. Alguns autores descrevem o problema como consistindo em encontrar um algoritmo distribuído para uma rede de processadores idênticos que, quando o algoritmo terminasse, um e apenas um dos processadores fosse designado líder e todos os outros soubessem que eles não são líderes. Para outros autores mais estritos é necessário ainda que todos os outros processadores não líderes tomem conhecimento da identificação do líder. O principal requisito para algoritmos de eleição de líder é de que a escolha do processador eleito seja única, ainda que vários nós estejam tentando eleger-se.

Outro ponto de diferenciação dos autores é em relação ao tratamento de falhas, que para alguns torna-se um fator inerente ao protocolo e para outros é de responsabilidade de outro nível do sistema. Neste trabalho a maioria dos algoritmos não consideram nenhum tipo de falha. São descritos alguns algoritmos que consideram falha do processador, mas que no caso de topologia em anel deve continuar enviando as mensagens recebidas para não quebrar a comunicação.

1.1.3 Relação com outros tipos de problemas

Os algoritmos de eleição de líder estão relacionados com outros problemas clássicos em computação distribuída: exclusão mútua, *membership*, nomeação em rede anônima, construção de uma árvore geradora para a rede.

O problema de exclusão mútua é definido como a propriedade de utilização mutuamente exclusiva de um recurso compartilhado. Para isso deve-se ter um algoritmo que escolha qual processo tem o direito de usar o recurso num dado instante. Assim como os algoritmos de exclusão mútua, a eleição de um líder deve escolher um processo único por vez. No entanto, ao contrário da exclusão mútua, a eleição de líder não necessariamente se preocupa com a justiça dessa escolha. Justiça refere-se ao fato de que um processo pode nunca ser escolhido como coordenador, mas, no caso de exclusão mútua, se um processo pede o uso do recurso, este deverá em algum momento obter o direito de uso.

Outro tipo de problema relacionado é o *membership problem*, que identifica quais processos estão ativos em um dado grupo. Neste problema, é conhecida a identificação de cada processo e deseja-se saber quais são os processos ativos no momento, ou seja, que nós fazem parte do grupo. Ao final do algoritmo, todos os processos do grupo devem conhecer quem faz parte do mesmo. Portanto, se todos possuem uma lista idêntica de processos ativos, para escolher um líder basta nomear o de menor, ou opcionalmente maior, identificação na lista.

Ainda uma outra classe de problemas associada com eleição é nomeação em rede anônima, ou seja uma rede onde os processadores não possuem identificadores, que consiste em identificar de forma única cada processador da rede. Utilizando um algoritmo de nomeação, torna-se fácil a tarefa de se escolher um coordenador; é apenas necessário, após a nomeação, tomar como líder o processo ao qual foi atribuída, por exemplo, a menor identificação.

O problema de eleição de líder em rede genérica está muito relacionado ao problema de se encontrar uma árvore geradora da rede através de um algoritmo decentralizado. O problema de construir uma árvore geradora de uma rede consiste em estabelecer um subconjunto de canais de comunicação que forma uma árvore geradora deste grafo representativo da rede [Bar96, GHS83, LR86, Tsi93]. Ao se estabelecer uma árvore geradora da rede, o processador raiz da árvore pode ser escolhido como líder.

1.2 Estrutura da dissertação

No capítulo 2 são apresentados alguns algoritmos clássicos para o problema de eleição de líder. São algoritmos mais antigos como [LeL77, CR79, Pet82, GM82] e com descrições encontradas na maior parte dos livros que abordam sistemas distribuídos como [Ray88, SG94, Tan92, CDK94].

O conceito de redes com uso de senso de direção é utilizado pelos algoritmos encontrados no capítulo 3. O fato de se ter uma orientação em relação aos canais de comunicação permite uma ordenação no processo de eleição. Neste capítulo são apresentados algoritmos para topologias completas e hipercubos.

O capítulo 4 apresenta algoritmos auto-estabilizantes, termo introduzido por Dijkstra em [Dij74]. Vários tipos de algoritmos, para diferentes topologias, anel, árvores, redes genéricas, vêm sendo desenvolvidos procurando diminuir o tempo necessário para a estabilização do sistema.

A introdução de probabilidade nos algoritmos para eleição é vista no capítulo 5. Alguns algoritmos que introduzem uma probabilidade de erro para conseguir quebrar a simetria da rede são apresentados. Esses algoritmos possuem uma pequena probabilidade de que ao terminar o algoritmo tenha-se nenhum ou mais de um líder. Caso isso aconteça uma nova eleição deve ser inicializada. Outro tipo de algoritmos que usam a probabilidade para que todos os participantes tenha a mesma chance de vencer também são apresentados.

Finalmente, no capítulo 6 são expostas as conclusões na forma de comentários sobre as comparações entre os algoritmos descritos.

Capítulo 2

Algoritmos Clássicos

Neste capítulo serão apresentados alguns algoritmos clássicos para solucionar o problema de eleição de líder. Os três primeiros algoritmos descritos ([LeL77, CR79, Pet82]) consideram que os processadores estão organizados em um anel unidirecional; assim, cada processador pode enviar mensagens a apenas um de seus vizinhos, obedecendo à ordem estabelecida no anel (horária ou anti-horária). O algoritmo de [GM82] assume que todos os processadores podem comunicar-se entre si.

2.1 Algoritmo de LeLann (1977)

Para esta solução cada processador possui uma identificação única que não é de conhecimento de todos os outros no anel. A principal estrutura de dados deste algoritmo [LeL77] é uma lista que contém a identificação dos processadores candidatos a líder no momento. O algoritmo funciona como segue para o processador P_i:

- P_i, ao detectar a falha do líder, cria uma nova lista de candidatos, inicialmente vazia. P_i envia a mensagem *ELEIÇÃO(i)* ao seu vizinho e adiciona o número i à sua lista de candidatos.
- Ao receber uma mensagem *ELEIÇÃO(j)*, o processo P_i deve executar uma das seguintes tarefas:
  1. Se esta é a primeira mensagem *ELEIÇÃO* de que ele tem conhecimento (recebeu ou enviou), P_i cria uma nova lista com i e j. Ele envia a mensagem *ELEIÇÃO(i)*, seguida de *ELEIÇÃO(j)*.
  2. Se $i \neq j$, P_i adiciona j a sua lista de candidatos e passa a mensagem para seu vizinho.
  3. Se $i = j$, então P_i contém a lista completa do todos os processos candidatos no sistema. P_i pode determinar o processo de maior identificação da lista, o qual deverá ser o novo líder.

```
Algoritmo de LeLann para o Processador i
Variáveis:
    viz; // canal do processador vizinho
    fim = FALSE; //terminou ou não a eleição
    Candidatos = cria(Candidatos); // lista de candidatos
    líder = -1; // identificação do líder

    if decide eleger-se then begin
        limpa(Candidatos);
        insere(i, Candidatos);
        send ELEIÇÃO(i) to viz;
    end
    while not fim do begin
        receive ELEIÇÃO(j) from c;
        if i ≠ j then begin
            insere(j, Candidatos);
            send ELEIÇÃO(j) to viz;
        end
        else begin
            líder = max(Candidatos);
            fim = TRUE;
        end
    end
```

Figura 2.1: Algoritmo de LeLann para anel unidirecional.

A figura 2.1 apresenta o algoritmo de LeLann detalhadamente. No algoritmo são usadas as seguintes variáveis:

- *viz*: endereço do canal que liga o processador a seu vizinho;
- *fim*: variável lógica que indica se o processador já terminou ou não a eleição;
- *Candidatos*: lista de processos candidatos;
- *líder*: indica quem é o líder atual.

As funções usadas no algoritmo são:

- *cria(X)*: cria uma lista com identificação X vazia;
- *limpa(X)*: esvazia a lista X;
- *insere(a, X)*: insere um elemento a na lista X;
- *max(X)*: devolve o valor máximo encontrado na lista X.

Na figura 2.2 é mostrado um exemplo do funcionamento do algoritmo de LeLann, onde Ci corresponde à lista de processadores candidatos do processador i. No estágio A da figura, o processador 1 decide eleger-se e envia uma mensagem *ELEIÇÃO* a seu vizinho, iniciando a eleição

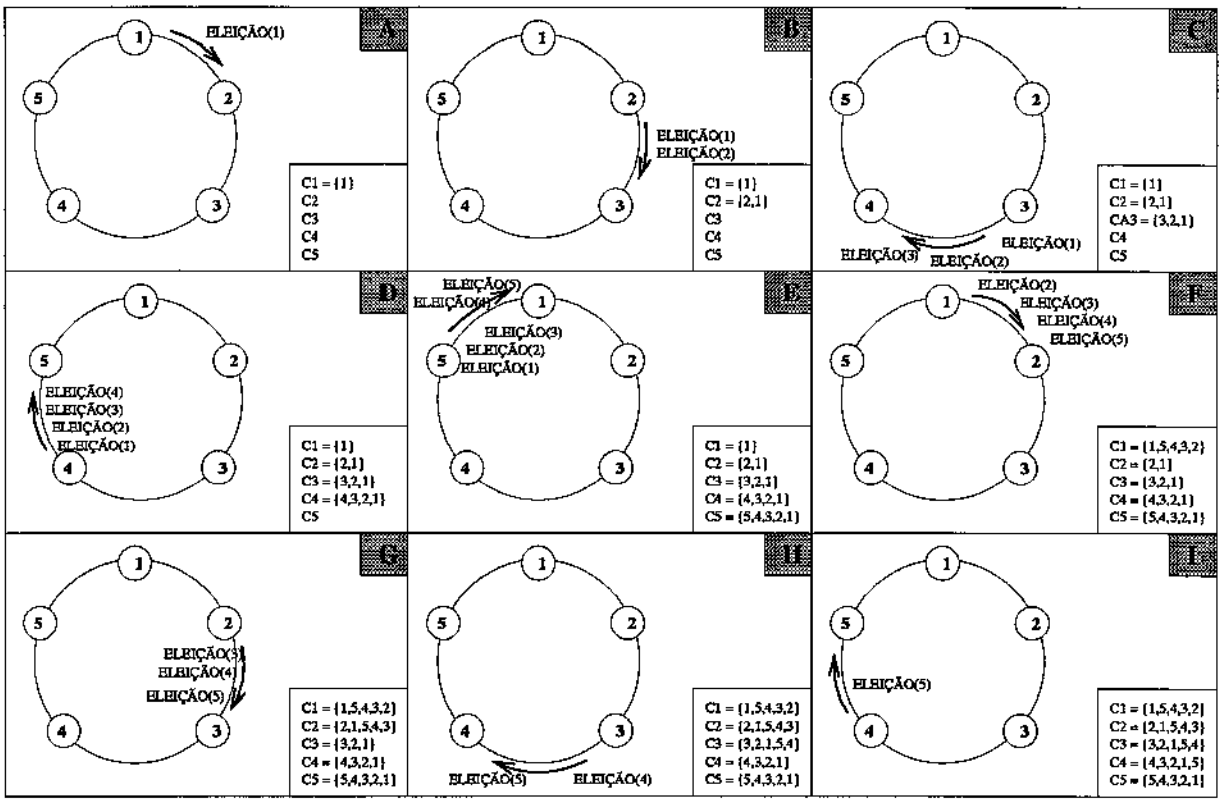


Figura 2.2: Exemplo de funcionamento do algoritmo de LeLann.

no anel. Cada processador ao tomar conhecimento de que a eleição está ocorrendo, normalmente ao receber uma mensagem *ELEIÇÃO*, inicializa sua lista de processadores candidatos com sua identificação e transmite sua mensagem *ELEIÇÃO*. Cada mensagem recebida por um processador é retransmitida ao seu vizinho no anel. A eleição termina quando o último processo completa sua lista de candidatos o que, no exemplo, é representado pela recepção da mensagem *ELEIÇÃO(5)* em trânsito no estágio I da figura. Ao final, como todos possuem a lista de candidatos com os mesmos processadores, todos sabem que o processador eleito foi o 5, e nenhuma mensagem adicional precisa ser enviada.

Se for assegurado que todas as mensagens circulam pelo anel chegando a seus destinos, pode-se afirmar que o algoritmo terminará elegendo um e apenas um líder. Ao ocorrer falha de um processador não se deve interromper o anel, ou seja, o sistema de transmissão de mensagem deve passar a mensagem, cujo destino é o nó que falhou, para o próximo nó no anel vizinho ao da falha.

Este algoritmo possui complexidade de mensagens igual a $O(n^2)$ mensagens, onde n é o número de processadores no anel, pois cada um dos processadores envia uma mensagem que deverá percorrer todos os processadores do anel.

A complexidade de tempo do algoritmo é igual a $O(n)$, visto que $n-1$ unidades de tempo após o primeiro processador iniciar a execução do algoritmo todos os processadores já iniciaram a eleição sendo necessárias mais n unidades de tempo para a identificação do último processador a iniciar a eleição percorra todo anel, totalizando $2n-1$ unidades de tempo.

2.2 Algoritmo de Chang e Roberts (1979)

Este algoritmo [CR79] foi um aperfeiçoamento da idéia de LeLann. Chang e Roberts procuraram desenvolver uma solução com uma complexidade de mensagens menor, mas para o mesmo modelo de sistema (anel unidirecional).

No algoritmo cada processador é inicialmente marcado como não candidato à eleição. Qualquer processador pode começar uma eleição (normalmente quando o processador detecta a falha do líder) e para isso deve primeiramente rotular-se como candidato, colocar sua identificação em uma mensagem *ELEIÇÃO* e enviá-la ao seu vizinho.

Quando um processador recebe uma mensagem de *ELEIÇÃO* ele compara sua identificação com a da mensagem. Se a identificação da mensagem for maior, então o processador retransmite a mesma mensagem a seu vizinho. Caso contrário, se ele não for ainda candidato, ele inicia sua eleição transmitindo a mensagem *ELEIÇÃO* com sua identificação. Ao passar uma mensagem *ELEIÇÃO* o processador marca-se como candidato.

Se a identificação da mensagem recebida for igual à do processador, ele torna-se líder. O líder rotula-se como não participante e manda uma mensagem *ELEITO* para seu vizinho notificando a identificação do novo líder. Quando um processador que não é líder recebe uma mensagem *ELEITO*, ele rotula-se como não participante e passa a mensagem adiante, tomando conhecimento de quem é o novo líder.

O algoritmo é descrito mais detalhadamente na figura 2.3. No algoritmo são usadas as seguintes variáveis

- *viz*: endereço do canal que liga o processador a seu vizinho;
- *candidato*: indica se o processador está ou não tentando eleger-se;
- *líder*: contém a identificação do líder atual.

A figura 2.4 descreve um exemplo de funcionamento do algoritmo de Chang e Roberts. Os processadores destacados com um círculo mais espesso representam os candidatos à eleição. No estágio A do exemplo a eleição é iniciada por dois processadores (1 e 3), que enviam mensagens *ELEIÇÃO* a seus vizinhos. Quando um processador i recebe uma mensagem *ELEIÇÃO(j)* ele compara i e j e decide se deve enviar uma mensagem e com qual identificação. Por exemplo, no estágio B da figura o processador 3 recebe uma mensagem *ELEIÇÃO(2)*; como $2 < 3$ e 3 já é candidato, o processador 3 não envia nenhuma mensagem. No mesmo estágio, o processador 4 recebe a mensagem *ELEIÇÃO(3)*, como $4 > 3$ e 4 ainda não era candidato, 4 envia a mensagem *ELEIÇÃO(4)*. Quando um processador receber uma mensagem *ELEIÇÃO* com sua

```
Algoritmo de Chang e Roberts para o Processador i
Variáveis:
    viz; // canal do processador vizinho
    fim = FALSE; //terminou ou não a eleição
    líder = -1; // identificação do líder
    candidato = FALSE;

    if decide eleger-se then begin
        candidato = TRUE;
        send ELEIÇÃO(i) to viz;
    end
    while not fim then begin
        receive M from c;
        if M = ELEIÇÃO(j) then begin
            if i = j then begin
                líder = i;
                candidato = FALSE;
                send ELEITO(i) to viz;
            end
            else begin
                if j > i then
                    send ELEIÇÃO(j) to viz;
                else if not candidato then begin
                    candidato = TRUE;
                    send ELEIÇÃO(i) to viz;
                end
        end
        if M = ELEITO(j) then begin
            if i ≠ j then begin
                send ELEITO(j) to viz;
                líder = j;
                candidato = FALSE;
            end
            fim = TRUE;
        end
    end
```

Figura 2.3: Algoritmo de Chang e Roberts.

identificação este toma conhecimento que sua mensagem percorreu todo o anel e ele possui a maior identificação. Portanto, ao final (estágio H) o processador com maior identificação no anel será o líder, no caso o processador 5. Na figura não é mostrada a mensagem *ELEITO* que deverá percorrer o anel informando que o processador 5 foi eleito.

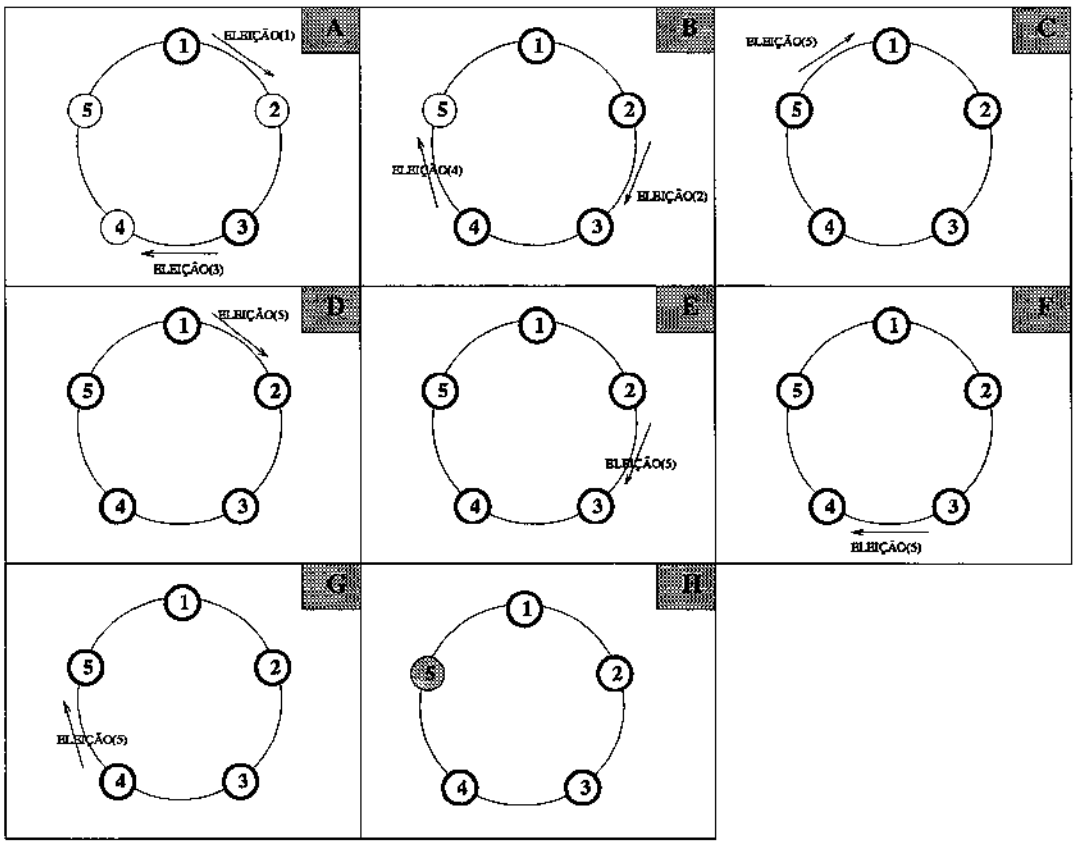


Figura 2.4: Um exemplo de funcionamento do algoritmo de Chang e Roberts.

Visto que o algoritmo compara as identificações dos processadores e apenas mantém as mensagens com identificação do maior processador, haverá um momento que a mensagem com a maior identificação terá passado por todos e o processador máximo será eleito.

O número de mensagens necessárias para a eleição depende do processador que inicia a eleição e da disposição dos identificadores dos processadores, que variam de 1 a n. Caso apenas um processador, P_i, inicie a eleição, o pior caso ocorre quando o processador P_j anterior a P_i, obedecendo a ordem do anel, possui a maior identificação. Neste caso, o algoritmo terá um custo de mensagens de $3n-1$ para a eleição de um novo líder ($n-1$ mensagens para ir de P_i a P_j, n mensagens para *ELEIÇÃO(j)* percorrer o anel e mais n mensagens *ELEITO*).

No artigo [CR79] é feita uma análise da complexidade de mensagens no caso médio. No cálculo são utilizadas as probabilidades para as diferentes disposições dos identificadores no

anel, e o resultado obtido é $O(n \log n)$, significando uma melhora em relação ao algoritmo de LeLann, onde a complexidade de mensagens é $O(n^2)$ para qualquer caso. No entanto, no pior caso, o algoritmo de Chang e Roberts mantém o custo de $O(n^2)$. A complexidade de tempo para o algoritmo de Chang e Roberts é a mesma do algoritmo de LeLann, ou seja, $O(n)$.

2.3 Algoritmo de Peterson (1982)

O algoritmo de Chang e Roberts apresentou uma melhoria no caso médio com custo de $O(n \log n)$. Hirschberg e Sinclair [HS80] propõem um algoritmo que requer, no pior caso, $O(n \log n)$ mensagens, mas utiliza uma topologia em anel bidirecional para comunicação. Hirschberg e Sinclair, no final de seu artigo, chegam a conjecturar que as soluções para anéis unidirecionais devem ser $\Omega(n^2)$. No entanto, Peterson [Pet82] apresenta um algoritmo que requer $O(n \log n)$ mensagens para eleição em anel unidirecional.

Peterson utiliza um esquema engenhoso, na verdade mostrando que um algoritmo para anel bidirecional pode ser executado em um anel unidirecional. As mensagens enviadas pelo algoritmo são: *ELEIÇÃO* e *ELEITO*. Inicialmente todos os processadores são participantes da eleição. Quando um processador deixa de ser participante não significa que este não é mais um candidato a líder, e sim que ele não está mais participando diretamente da detecção do processador com maior identificação. No entanto, antes de deixar de ser participante um processador enviou pelo menos uma mensagem *ELEIÇÃO* carregando sua identificação, ou seja, o anel já tomou conhecimento de sua identificação. Um processador não participante apenas retransmite as mensagens que recebe através do anel, até que receba uma mensagem *ELEIÇÃO* com sua identificação indicando que ele possui a maior identificação dentre os processadores do anel, ou que receba uma mensagem *ELEITO* indicando o novo líder.

O algoritmo trabalha em fases e o objetivo é reduzir a pelo menos metade o número de processadores participantes a cada fase executada. Cada processador possui um identificador temporário (*tid*) que é inicializado com sua própria identificação. A cada fase um processador P_i recebe uma mensagem com o *tid* do processador participante que o antecede no anel, P_j, e outra mensagem com o *tid* do processador que antecede P_j. Uma comparação entre os três *tid's* é feita. Se o *tid* de P_j for o maior dentre os três então P_i atualiza seu *tid* para o valor de *tid* de P_j e inicia uma nova fase. Caso contrário, P_i passa seu estado para não participante. Esta comparação entre os identificadores de três processadores consecutivos no anel simula um bidirecionamento. Em um anel bidirecional um processador continuaria participando da eleição caso ele possuísse um identificador maior que os de seus vizinhos da esquerda e da direita. Na figura 2.5 o algoritmo é descrito com mais detalhes. São usadas as seguintes variáveis:

- *tid, ntid e nntid*: armazenam os *id* temporários usados para a comparação entre os processadores;
- *participante*: indica se o processador está ou não participando da eleição;
- *líder*: contém a identificação do líder atual.

```
Algoritmo de Peterson para o Processador i
Variáveis:
    tid = i; // id temporário
    participante = TRUE; // participante da eleição
    viz; // canal do processador vizinho
    fim = FALSE; //terminou ou não a eleição
    líder = -1; // identificação do líder

    while not(fim) do begin
        if (participante) then begin
            send ELEIÇÃO(tid) to viz;
            receive M from c;
            if M = ELEIÇÃO(ntid) then begin
                if (ntid = i) then
                    send ELEITO(i) to viz;
                if (tid > ntid) then
                    send ELEIÇÃO(tid) to viz;
                else send ELEIÇÃO(ntid) to viz;
                receive M from c;
                if M = ELEIÇÃO(nntid) then begin
                    if (nntid = i) then
                        send ELEITO(i) to viz;
                    if (ntid ≥ max(tid, nntid)) then tid = ntid;
                    else participante = FALSE;
                end
            end
            else if M = ELEITO(tid) then begin
                if tid ≠ i then
                    send ELEITO(tid) to viz;
                líder = tid;
                fim = TRUE;
            end
        end
        if not (participante or fim) then begin
            receive M from c;
            if M = ELEIÇÃO(tid) and líder = -1 then
                if (tid = i) then
                    send ELEITO(i) to viz;
                else send ELEIÇÃO(tid) to viz;
            end
            else if M = ELEITO(tid) then begin
                if tid ≠ i then begin
                    send ELEITO(tid) to viz;
                    líder = tid;
                end
                fim = TRUE;
            end
        end
    end
```

Figura 2.5: Algoritmo de Peterson.

A figura 2.6 apresenta um exemplo de funcionamento do algoritmo de Peterson. Na figura os processadores estão representados por um círculo mais espesso. No estágio A todos os processadores são participantes e enviam mensagem $ELEIÇÃO$ a seus vizinhos. O envio da segunda mensagem da primeira fase é mostrado no estágio B da figura. Ao receber a segunda mensagem (estágio C) os processadores 3, 4 e 5 deixam de ser participantes e apenas 1 e 2 iniciam uma nova fase. No estágio F o processador 5 percebe que a mensagem $ELEIÇÃO(5)$ percorreu todo o anel, e inicia o anúncio de sua eleição, que termina no estágio L. Ao final todos os processadores têm conhecimento que 5 foi eleito.

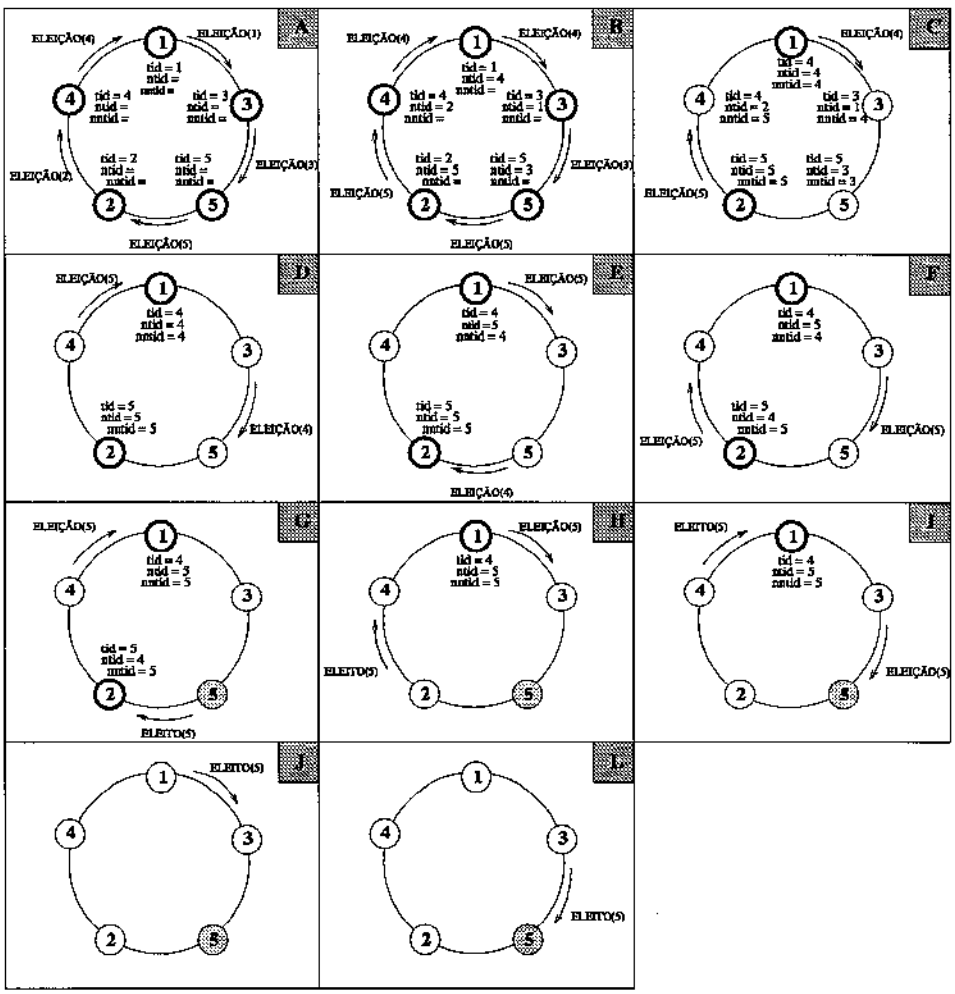


Figura 2.6: Um exemplo de funcionamento do algoritmo de Peterson.

Facilmente podemos notar que o número máximo de mensagens enviadas é $2n\lfloor \log n \rfloor + n$, pois com a eliminação de pelo menos metade dos participantes por fase, temos $\lfloor \log n \rfloor$ fases e a cada fase são enviadas $2n$ mensagens, sendo que as outras n mensagens servem para anunciar a

identificação do processo eleito a todos os outros do anel. O número de participantes dimunui pela metade a cada fase pois um processador só permanecerá participante se o processador participante a sua esquerda deixar de sê-lo. No artigo são mostrados ainda alguns melhoramentos que diminuem o número de mensagens transmitidas, mas ainda mantêm a complexidade de mensagens igual a $O(n \log n)$. A idéia é antecipar a passagem do processador para o estado não participante, não esperando a comparação como os dois vizinhos. A cada comparação com a identificação vinda do vizinho, o processador pode decidir deixar ou não de ser participante. A complexidade de tempo do algoritmo é $O(n \log n)$.

2.4 Algoritmo de Garcia-Molina (1982)

O algoritmo apresentado por Garcia-Molina em [GM82], conhecido também como algoritmo do Ditador [1], trabalha com os processadores organizados em uma rede completa. O princípio básico para dar início à eleição é que, ao detectar a falha do líder, o processador tenta eleger-se.

Quando um processador P_i envia uma mensagem ao líder fazendo um pedido e não obtém resposta durante um intervalo de tempo T, ele assume que o líder falhou e tenta eleger-se como novo líder. O processador P_i envia uma mensagem *ELEIÇÃO* para todo processador com identificação maior que a sua e espera por mensagens de um desses processadores durante um tempo T. Se nenhuma resposta for recebida, P_i assume que todos os processadores com identificação maior que a sua falharam, declara-se líder e envia mensagens a todos os processadores de identificação mais baixa, anunciando ser o novo líder.

No entanto, se alguma resposta for recebida, P_i espera durante um intervalo de tempo T por uma mensagem indicando que um processo com identificação maior que i é o novo líder. Se não for recebida resposta alguma em T, P_i assume que o processador de maior identificação falhou e tenta eleger-se novamente.

Durante a execução, se P_i não é o líder, este pode receber dois tipos de mensagens:

- uma mensagem indicando que P_j é o novo líder ($j > i$). Então o processador P_i armazena a informação do novo líder.
- uma mensagem indicando que P_j iniciou uma eleição ($j < i$). P_i deverá enviar uma resposta a P_j para que P_j tenha conhecimento de que há um processador com identificação maior que a sua ainda ativo, e tentará eleger-se.

Quando um processador se recupera de uma falha, ele imediatamente executa o algoritmo tentando sua eleição. Se todos os outros processadores ativos possuírem identificação menor, esse processador que acabou de se recuperar será o novo líder. O algoritmo é descrito com maiores detalhes na figura 2.7. As mensagens enviadas pelo algoritmo são: *ELEIÇÃO* (indicando que o processador que enviou a mensagem deseja eleger-se), *RESPOSTA* (é gerada quando um processador recebe uma mensagem *ELEIÇÃO*) e *ELEITO* (indica o novo líder). No algoritmo são usadas as seguintes variáveis:

[1] *bully algorithm*

- *estado*: pode possuir quatro valores: passivo (o processador não é líder e nem está querendo eleger-se), eleger-se (o processador acha que o líder falhou e vai tentar eleger-se), elegendo-se (o processador está tentando eleger-se) ou eleito (o processador é o líder).
- *líder*: indica quem é o líder atual.

As funções usadas no algoritmo são:

- *inicia_timeout()*: inicializa o contador de *timeout*;
- *desativa_timeout()*: desativa o contador de *timeout*;
- *timeout(t)*: retorna o valor verdadeiro se o contador de *timeout* chegar ao valor t.

Um exemplo do funcionamento do algoritmo pode ser visualizado na figura 2.8. Considere 5 processadores com identificadores de 1 a 5. O processador líder (5) falha; suponha que o processador 2 descobre que 5 falhou. O processador 2 envia mensagens *ELEIÇÃO* a todos com identificação maior que a sua (estágio 1 da figura). Os processadores 3 e 4 respondem a 2 e iniciam suas eleições (estágios 2 e 3). O processador 4 envia resposta a 3 e falha (estágio 4). Expira, então, o tempo e 2 tenta eleger-se novamente (estágio 4), mas, quem acaba elegendo-se é o processador 3 (estágios 5 e 6).

Assegurando que as mensagens enviadas sempre chegam a seus destinos, ou seja, que não ocorram falhas na rede de comunicação, o processador ativo com maior identificação será eleito. Ao analisarmos o algoritmo, é fácil notar que sua complexidade de mensagens é $O(n^2)$, o que, no pior caso, corresponde ao processador de mais baixa identificação iniciar sua eleição sendo que os demais processadores ainda estão ativos. A complexidade de tempo necessária para que um processador seja eleito é igual a $O(1)$ considerando somente a falha de processadores e não do meio de comunicação.

2.5 Alguns Limites Inferiores

Alguns resultados para os limites inferiores de complexidade de mensagens para o problema de eleição de líder são apresentados na literatura. A tabela a seguir mostra alguns desses limites inferiores que foram determinados de acordo com certos modelos de rede. As referências apresentadas na tabela demonstram a obtenção dos limites inferiores.

```
Algoritmo de Garcia-Molina para o Processador i
Variáveis:
    fim = FALSE; //terminou ou não a eleição
    líder = -1; // identificação do líder
    estado = passivo; // estado processador (passivo, eleger-se, elegendo-se, eleito)

    if decide eleger-se then
        estado = eleger-se;
    while not fim do begin
        if estado = eleger-se then begin
            for ( j = i + 1; j < n; j = j + 1) do
                send ELEIÇÃO to j;
            estado = elegendo-se;
            inicia_timeout();
        end
        receive M from j;
        if M ≠ nil then begin
            case M of
                RESPOSTA:
                    inicia_timeout();
                    estado = passivo;
                ELEIÇÃO:
                    send RESPOSTA to j;
                    if estado ≠ elegendo-se then begin
                        estado = eleger-se;
                ELEITO:
                    líder = j;
                    desativa_timeout();
                    fim = TRUE;
        end
        if timeout(T) then begin
            if estado = elegendo-se then begin
                for ( j = 1; j < i; j = j + 1) do
                    send ELEITO to j;
                estado = eleito;
                fim = TRUE;
            end
            else
                estado = eleger-se;
        end
    end
```

Figura 2.7: Algoritmo de Garcia-Molina.

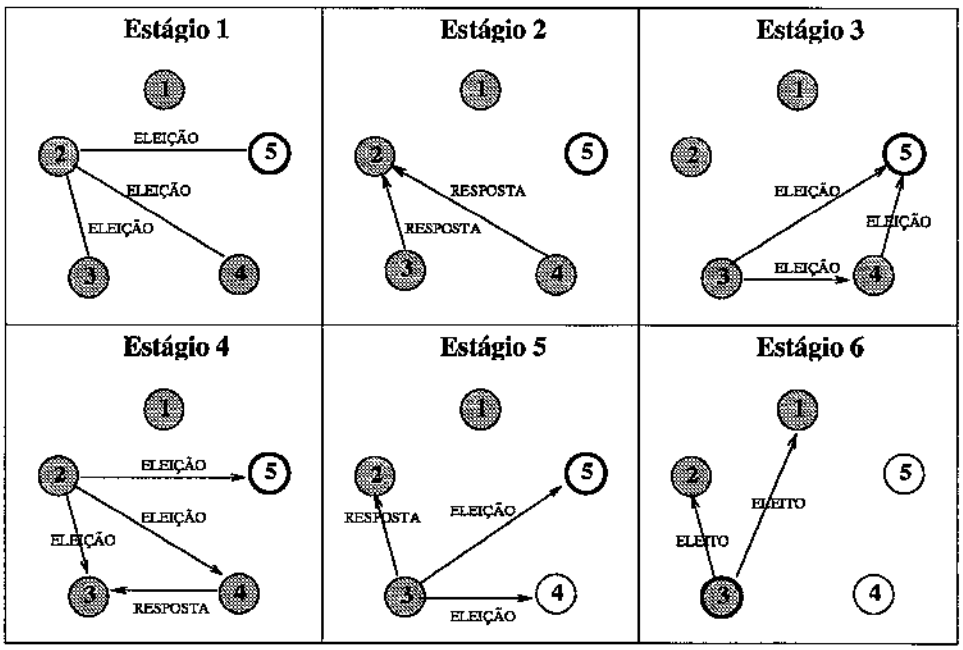


Figura 2.8: Um exemplo de funcionamento do algoritmo de Garcia-Molina.

Limite	modelo	Referência
$\Omega(n \log n)$	rede completa síncrona	[AG91]
$\Omega(n \log n)$	rede completa assíncrona	[KMZ84]
$\Omega(n \log n)$	anel unidirecional	[PKR84]
$\Omega(n \log n)$	anel bidirecional	[Lyn96]
$\Omega(n \log n + \|E\|)$	redes genéricas	[Tel94]

Tabela 2.1: Tabela com alguns limites inferiores para complexidade de mensagens para o problema de eleição de líder.

Capítulo 3

Uso de Senso de Direção

Dizemos que uma rede possui *senso de direção* se os processadores são capazes de fazer distinção entre seus canais de comunicação, de acordo com um esquema globalmente consistente. Na figura 3.1 temos um exemplo de rede com senso de direção, onde as arestas direcionadas representam um ciclo Hamiltoniano e os rótulos dos canais de comunicação representam a distância entre os processadores através do ciclo.

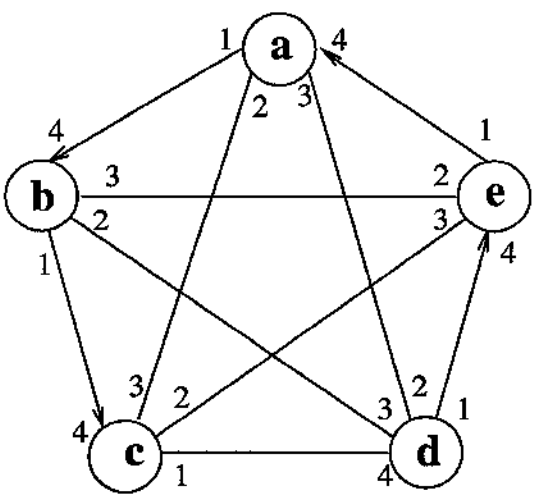


Figura 3.1: Exemplo de uma rede completa com senso de direção.

Neste capítulo estudaremos dois algoritmos para eleição em redes completas e outros dois algoritmos que trabalham com redes em hipercubo. O senso de direção é usado para diminuir a número de mensagens necessárias para realizar a eleição.

3.1 Eleição em Rede Completa

No caso de redes completas, senso de direção refere-se ao conhecimento de um ciclo Hamiltoniano e à existência de rótulos em cada canal de comunicação dos processadores, os quais são baseados na distância entre os processadores percorrendo-se o ciclo Hamiltoniano como no exemplo da figura 3.1. Neste caso, se um processador i está a uma distância d de j, então j está a uma distância $n - d$ de i (n é o número total de processadores).

A seguir serão apresentados dois algoritmos para eleição de líder em redes assíncronas completas com senso de direção. O primeiro algoritmo [LMW86], mais antigo e que se assemelha ao algoritmo de [Pet82], foi um dos primeiros a propor o uso de senso de direção para diminuir o número de mensagens. Em [Sin97] é proposta uma nova forma de fazer uso do senso de direção a fim de tornar a eleição mais eficiente. Os dois possuem o mesmo número de mensagens, $O(n)$, mas diferem na complexidade de tempo, o primeiro com $O(n)$ e o segundo com $O(\log n)$. Como citado em [LMW86], para o caso de uma rede sem senso de direção são necessárias $\Omega(n \log n)$ mensagens para resolver o problema de eleição.

3.1.1 Algoritmo de Loui et al.

O algoritmo de [LMW86] apresentado nesta seção considera redes completas assíncronas com senso de direção, onde cada processador possui um identificador único, fazendo uso de um ciclo Hamiltoniano definido na rede através de rótulos nos canais de comunicação.

Inicialmente todos os processadores são *ativos* até que todos, menos o de maior identificação, tornam-se *passivos*. Um processador *ativo* i, ao receber uma mensagem com uma identificação j tal que $j > i$, passa seu estado para *passivo*. Quando $i > j$, i envia uma mensagem para que j torne-se passivo. Cada mensagem contém a distância de i ao processador j. Os processadores usam a distância para determinar qual canal de comunicação devem usar.

Os processadores podem ter três valores para seu estado: *ativo*, *passivo* e *eleito*. O algoritmo inicia com todos os processadores enviando a mensagem $ELEIÇÃO(id, d)$ aos processadores com distância 1, sendo que id é a identificação do originador da mensagem e d representa a que distância o originador está do receptor da mensagem. Se o processador i está *ativo* ao receber a mensagem $ELEIÇÃO(j, e)$ e $j < i$, então i envia uma mensagem $ELEIÇÃO$ diretamente para j, que está a distância $n - e$ de i, para assegurar que j passe seu estado para *passivo* e saia da disputa. Caso i esteja *ativo* e $j > i$, então i torna-se *passivo*. Caso i já seja *passivo*, então i transmite a mensagem $ELEIÇÃO(j, e')$ ao processador com distância $n - d$ (no caso de um processador *passivo*, d representa a sua distância em relação ao processador que o desativou), onde $e' = (n - d + e) \bmod n$. A transmissão de $ELEIÇÃO(j, e')$ de um processador *passivo* i para um processador *ativo* k faz com que k comunique-se diretamente com j. Assim nenhum processador enviará mais mensagens a i durante a eleição.

Para assegurar o número de mensagens apresentado em [LMW86], foi publicada uma correção posterior onde foi indicado que o algoritmo deveria operar em fases. Deve-se então incluir o número de fase na mensagem e os processadores devem receber as mensagens obedecendo à

ordem crescente do número de fase. Na figura 3.2 o algoritmo é apresentado.

Na figura 3.3 é apresentado um exemplo de funcionamento do algoritmo de Loui et al., onde temos 7 processadores com identificadores variando em {1,2,...,7}. Os processadores ativos são apresentados em cor escura e quando um processador torna-se passivo passamos a desenhá-lo na cor branca. A cada fase as mensagens enviadas são representadas pelas setas entre os processadores e o número ao lado de cada seta representa o identificador que a mensagem contém. Na primeira fase todos os processadores enviam mensagens a seus vizinhos com distância 1. Na fase seguinte os processadores que receberam mensagem com identificação maior que a sua tornam-se passivos. Os demais enviam uma mensagem para desativar o originador da mensagem que receberam, que é um processador com identificação menor. A troca de mensagens continua até que um processador ativo receba uma mensagem com identificação igual à sua. Este deverá então declarar-se líder.

Sem perda de generalidade assumimos que todos os processadores ativos enviam suas mensagens simultaneamente no início de cada fase. Alguns processadores passivos também enviam mensagens. A fase p termina quando todos os processadores ativos recebem as mensagens enviadas durante a fase p. Considere que as fases são numerados de 1 a p, sendo que p começa quando $p-1$ termina. Seja c_p o número de processadores ativos no início da fase p ($c_0 = n$). Sabemos que cada processador envia apenas uma única mensagem após tornar-se passivo, totalizando (n-1) mensagens enviadas por processadores passivos durante todo o algoritmo. Portanto o número total de mensagens enviadas é:

$$(n - 1 + \sum_{p \geq 0} c_p).$$

Em [LMW86] foi demonstrado por indução no número de termos do somatório que, considerando $\Phi = \frac{1}{2}(1 + \sqrt{5})$, temos:

$$\sum_{p \geq 0} c_p < \Phi^2 n.$$

Logo,

$$(n - 1 + \sum_{p \geq 0} c_p) < n - 1 + \Phi^2 n < 3.62n.$$

Assim, a complexidade de mensagens do algoritmo é $O(n)$, e considerando que o número de bits necessários para a identificação dos processadores seja $O(\log n)$, a complexidade de mensagens em bits é $O(n \log n)$.

Toda mensagem enviada durante uma fase deve chegar a um processador ativo nesta mesma fase, no entanto pode ser necessário que processadores passivos retransmitam mensagens que receberam durante a fase. Portanto para calcular a complexidade de tempo não basta saber o número de fases, mas deve-se contar o tempo de cada fase, que é determinado pelo número de retransmissões. Sabemos que uma mensagem para chegar a um processador ativo pode demorar no máximo 1 mais o número de processadores que se tornaram passivos na fase anterior, ou seja, a mensagem no máximo percorreria todos os processadores que se tornaram passivos na

Algoritmo de Loui et al. para o Processador i
Variáveis:

```
d = n - 1;
estado = ativo;
fase = 1;
fim = FALSE;

while not fim do
begin
    if estado = ativo then
    begin
        send ELEIÇÃO(i, n − d, fase) to n − d;
        receive ELEIÇÃO(j,d,f) from e;
        while f ≠ fase do
            recoloca a mensagem na fila de mensagens a receber e recebe outra mensagem
        fase = fase +1;
        if i < j then
            estado = passivo;
        else if i = j then
            begin
                estado = eleito;
                fim = TRUE;
                for (e = 1; e < n; e++)
                    send ELEITO(i) to e;
            end
    end
    if estado = passivo then
    begin
        receive M from e;
        fase = fase +1;
        if M = ELEIÇÃO(j,d',f) then begin
            send ELEIÇÃO(j, (n − d + d') mod n, fase) to n − d;
        end else if M = ELEITO(j) then
            fim = TRUE;
    end
end
```

Figura 3.2: Algoritmo de Loui et al. para redes completas com senso de direção.

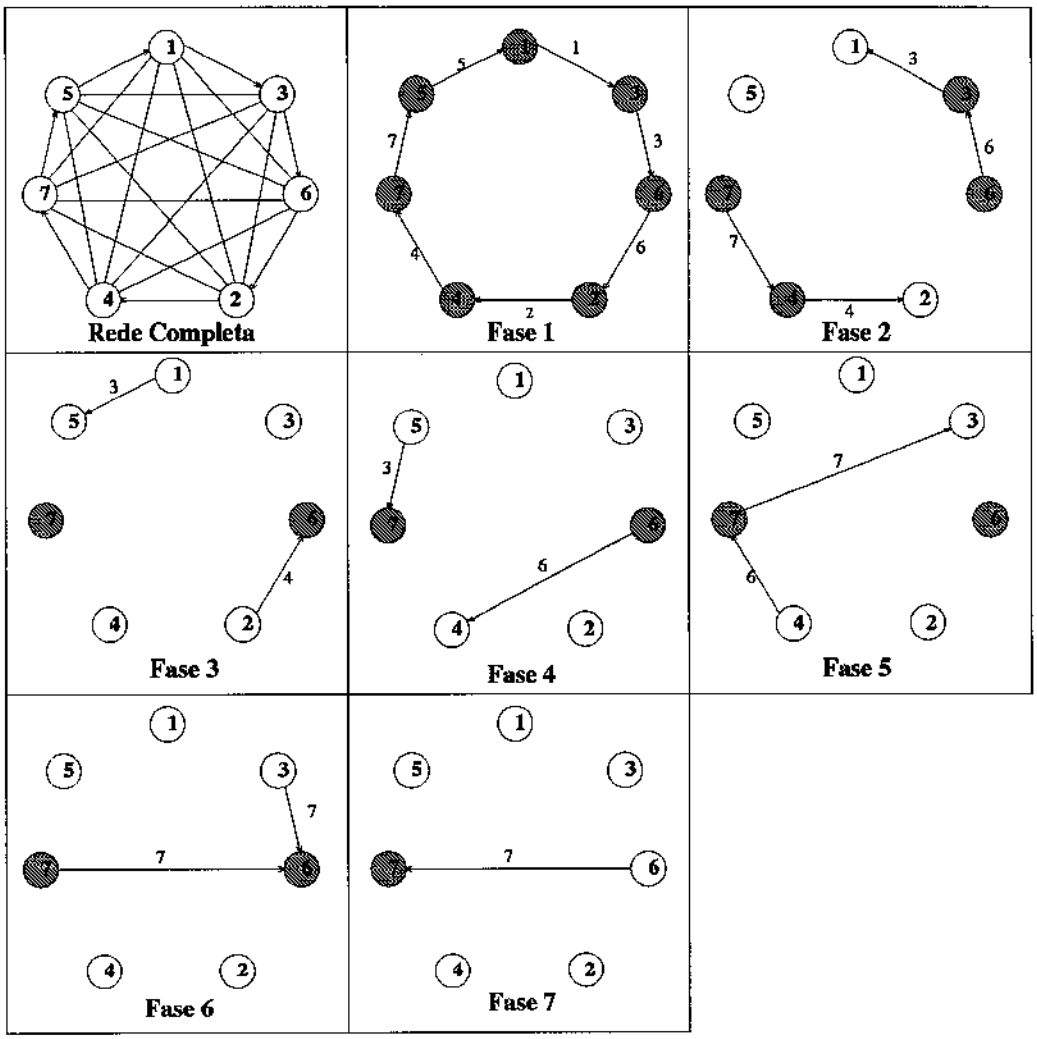


Figura 3.3: Exemplo de funcionamento do algoritmo de Loui et al. para redes completas com senso de direção.

fase anterior até chegar a um processador ativo. Logo a complexidade de tempo é dada por:

$$\sum_{p\geq 0} tempo\ da\ fase\ p = \sum_{p\geq 0} 1 + (n - c_p) = p + (n-1).$$

Como p (número de fases) é $\log_\Phi n + O(1)$, temos que a complexidade de tempo é $O(n + \log_\Phi n)$, ou seja, $O(n)$.

3.1.2 Algoritmo de Singh

Assim como o algoritmo apresentado anteriormente, no artigo de Singh [Sin97] é proposto um algoritmo para eleição em redes assíncronas completas usando senso de direção, onde cada processador possui um identificador único. O protocolo possui duas fases. Na primeira o número de

processadores ativos é reduzido para aproximadamente $n/\log n$, onde n corresponde ao número de processadores da rede. Para isso o anel lógico, formado pelo ciclo Hamiltoniano, é divido em k sub-anéis, onde $k = n/2^{\lceil \log \log n \rceil}$. Ao final da primeira fase há no máximo um processador ativo em cada sub-anel. Na segunda fase do protocolo, os processadores ativos restantes competem entre si para elegerem um único líder.

A relação $(\ell, d) < (\ell', d')$ é considerada verdadeira caso $(\ell < \ell')$ ou $((\ell = \ell')e(d < d'))$. Usaremos a notação $i[d]$ representando o nó com distância d de i ($i[0] = i$) e $i[x..y]$ representando o conjunto $\{i[x], i[x+1], ..., i[y]\}$. Por exemplo, de acordo com a figura 3.1, $c[3] = a$ e $c[1..3] = \{d, e, a\}$. É considerado que n é potência de 2. Posteriormente serão comentadas as modificações que devem ser feitas quando n não é potência de 2. Os processadores são organizados logicamente em um anel que corresponde ao ciclo Hamiltoniano. Os sub-anéis são formados da seguinte maneira: o anel a que o processador i pertence é composto pelo conjunto $\{i[0], i[k], i[2k], ..., i[n-k]\}$ de processadores. Na figura 3.4 temos um exemplo onde $n = 16$ e $k = 4$, sendo que os sub-anéis são: $\{a, e, i, m\}$, $\{b, f, j, n\}$, $\{c, g, k, o\}$ e $\{d, h, l, p\}$ representados por diferentes cores. Será usada a notação A_i para indicar o sub-anel a que i pertence.

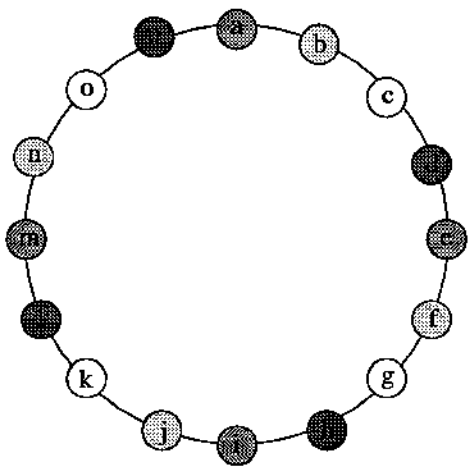


Figura 3.4: Exemplo de rede dividida em 4 sub-anéis.

Abaixo temos uma descrição das variáveis usadas no algoritmo para um processador i:

- *estado*: Pode ter três valores: *passivo* (inicialmente e quando um processador o desativa), *ativo* (enquanto um processador continua competindo na eleição) e *eleito* (caso i seja o líder no final do algoritmo).
- *nível*: Inicia com zero e posteriormente indica quantos processadores i desativou na primeira fase.
- *passo*: Inicialmente possui o valor zero. Indica em que passo da segunda fase i está.

- *líder*: Se *estado* = *ativo*, *líder*= 0. Caso i seja desativado, *líder* indicará o processador que continua sendo ativo em A_i ao final da primeira fase.

- $fase$: Indica a fase do algoritmo em que o processador está.

Na primeira fase a eleição ocorre separadamente em cada sub-anel. Ao acordar um processador passa seu estado para ativo e tenta desativar seqüencialmente os demais processadores de seu sub-anel. Para desativar um processador, i deve enviar uma mensagem $ELEIÇÃO(fase,nível,i)$ ao processador $j = i[k*nível]$ e esperar o recebimento de uma mensagem $ACEITO(fase,nível)$. Um processador usa seu nível e sua identificação para competir com outros processadores. Um processador j é desativado caso receba uma mensagem $ELEIÇÃO(fase_i,nível_i,i)$ de um processador i e $(nível_j,j) < (nível_i,i)$. Se um processador ativo ao receber uma mensagem $ELEIÇÃO$ não satisfizer a condição anterior este deve ignorar a mensagem e o processador que a enviou ficará bloqueado. Caso um processador j, executando a primeira fase do protocolo, receba uma mensagem $ELEIÇÃO$ de outro que já esteja na segunda fase, então j passará seu estado para passivo e enviará a mensagem $ACEITO$. Um processador i, na primeira fase, ao desativar j deve verificar qual o valor de $nível_j$ para que o próximo processador que i tente desativar seja $i[(nível_i+nível_j+1)*k]$, pois os processadores em $i[k*(nível_i+1),...,k*(nível_i+nível_j)]$ não estão mais ativos (foram desativados por j). Se $nível_i \geq n/k$ (i desativou todos os processadores de seu sub-anel) então i deve enviar a mensagem $LÍDER_ANEL(i)$ aos processadores de A_i informando que foi eleito na primeira fase. O processador i fica aguardando até que todos os processadores de A_i respondam com uma mensagem $CIENTE$ indicando o recebimento de sua mensagem, para então passar à segunda fase.

Na segunda fase temos k sub-anéis com no máximo um processador ativo por sub-anel. Esta fase possui $\log n$ passos. No primeiro passo um processador i envia uma mensagem para desativar um processador. No ℓ-ésimo passo, i envia mensagens para desativar $2^{\ell-1}$ processadores. Quando dois processadores competem nesta fase ganha quem tem maior par (*passo, identificação*). Ao iniciar a segunda fase ($fase = 2$) o processador i executa a atribuição $passo = 1$ e envia uma mensagem $ELEIÇÃO(fase,passo,i)$ ao processador $j = i[k/2]$. Caso j não seja o processador ativo de A_j, este deve enviar a mensagem *INFORMA(líder)* a i para que i tente desativar o *líder* de A_j. Caso i obtenha sucesso ao tentar desativar o processador de $A_{i[k/2]}$, este incrementa a variável *passo* e tenta desativar os líderes de $A_{i[k/4]}$ e $A_{i[3k/4]}$. Generalizando, no $passo = \ell$, i tenta desativar os processadores ativos dos sub-anéis $\{A_{i[k/2^\ell]}, A_{i[3k/2^\ell]}, ..., A_{i[(2^\ell-1)k/2^\ell]}\}$. Se i conseguir desativar todos os processadores para os quais enviou mensagens no passo ℓ, então ele incremente a variável *passo*. Caso $passo > \log k$ então i pode declarar-se eleito enviando uma mensagem $ELEITO(i)$ a todos os processadores da rede. As figuras 3.5, 3.6 e 3.7 apresentam o algoritmo com mais detalhes.

É interessante notar que quando $passo = 1$ o processador i tenta desativar o líder de $A_{i[k/2]}$ (suponha que seja j). O processador j por sua vez, ao entrar na segunda fase, tenta desativar o processador ativo de $A_{j[k/2]}$, que corresponde a i. Portanto, quando i tenta desativar j, j também tenta desativar i. Assim apenas um dos dois irá continuar tentando se eleger. Ou seja quando todos os processadores executarem o primeiro passo da segunda fase restarão no

```
Algoritmo de Singh para o Processador i
Variáveis:
    estado = ativo;
    nível = 0;
    fase = 1;
    líder = 0;

    while estado = ativo and fase = 1 do begin
        if nível > n/k then begin
            send LÍDER_ANEL(i) to all j ∈ A_i;
            respostas = 0;
        end else send ELEIÇÃO(1, nível, i) to i[(nível+1) * k];
        recebido = FALSE;
        while estado = ativo and fase = 1 and not recebido do begin
            receive M from i[d];
            case M of
                ELEIÇÃO(1, n, j): if (nível, i) < (n, j) then begin
                        send ACEITO(1,nível) to i[d];
                        estado = passivo;
                    end
                ACEITO(1,n): nível = nível +n + 1;
                    recebido = TRUE;
                CIENTE(i): respostas = respostas + 1;
                    if respostas = n/k − 1 then fase = 2;
                ELEIÇÃO(2, n, j): estado = passivo;
                    fase = 2;
                    send ACEITO(2,nível) to i[d];
                ELEITO(j): estado = passivo;
                    líder = j;
        end
    end
```

Figura 3.5: Algoritmo de Singh para redes completas com senso de direção (parte 1).

```
if estado = ativo and fase = 2 then begin
    passo = 0;
    while estado = ativo do begin
        for (m = 1; m ≤ 2^passo − 1; m = m + 2)
            send ELEIÇÃO(2, passo, i) to i[m * k/2^passo];
        recebido = FALSE;
        respostas = 0;
        while estado = ativo and not recebido do begin
            receive M from i[d];
            case M of
                ELEIÇÃO(1, n, j): skip;
                ELEIÇÃO(2, n, j): if (passo, i) < (n, j) then begin
                        send ACEITO(2, passo) to i[d];
                        estado = passivo;
                    end
                INFORME(x): send ELEIÇÃO(2, passo, i) to i[d + x];
                ACEITO(2,n): respostas = respostas + 1;
                    if respostas = 2^(passo−1) then begin
                        passo = passo + 1;
                        if passo = log k + 1 then begin
                            estado = eleito;
                            for (m = 1; m < n; m = m + 1) send ELEITO(i) to m;
                        end else recebido = TRUE;
                    end
        end
    end
end
```

Figura 3.6: Algoritmo de Singh para redes completas com senso de direção (parte 2).

```
fim = FALSE;
while estado = passivo and not fim do begin
      receive M from i[d];
      case M of
            ELEIÇÃO(1, n, j): if fase = 1 then send ACEITO(1,0) to i[d];
            ELEIÇÃO(2, n, j): if fase = 1 and líder = 0 then begin
                        send ACEITO(2, 0) to i[d];
                        fase = 2;
                  end else send INFORME(líder) to i[d];
            LÍDER_ANEL(j): if fase = 1 then begin
                        líder = j;
                        fase = 2;
                        send CIENTE(j) to i[d];
                  end
            ELEITO(j): líder = j;
                  fim = TRUE;
            ACEITO(1, n): skip;
            ACEITO(2, n): skip;
            CIENTE(i): skip;
            INFORME(j): skip;
end
```

Figura 3.7: Algoritmo de Singh para redes completas com senso de direção (parte 3).

máximo $k/2$ processadores ativos. Se i não for desativado no primeiro passo, deverá enviar mensagens para desativar $A_{i[k/4]}$ e $A_{i[3k/4]}$ no segundo passo. Quando uns dos processadores ativos de $A_{i[k/4]}$ ou $A_{i[3k/4]}$ (haverá apenas um ativo ou em $A_{i[k/4]}$ ou em $A_{i[3k/4]}$) for executar o segundo passo este deverá enviar mensagem de *ELEIÇÃO* a i e ao processador ativo de $A_{i[k/2]}$. Portanto, novamente, os processadores irão competir um contra o outro. Com este método, no máximo $k/2^{\ell-1}$ processadores ativos entram no passo ℓ.

Considerando $des(\ell, i) = \{i[k], i[2k], ..., i[\ell * k]\}$ temos que, a qualquer momento, se *nível*> 0 e $fase = 1$ então não existe nenhum processador ativo em $des(nível, i)$. Admitindo que o algoritmo inicie no tempo 0 e que todas as mensagens levem uma unidade de tempo para serem entregues, é necessário tempo $O(n/k)$ para que o primeiro processador a acordar entre na segunda fase. Como visto, na segunda fase teremos no máximo $k/2^{\ell-1}$ processadores ativos iniciando o passo ℓ, ou seja, no passo $\log k$ restará apenas um processador ativo. Portanto, o tempo transcorrido entre um dos processadores entrar na segunda fase e algum deles se declarar líder é $O(\log k)$. Logo, a complexidade de tempo do algoritmo é $O(\log n)$.

As mensagens enviadas na primeira fase são ELEIÇÃO, ACEITO, LÍDER_ANEL e CIENTE, sendo necessário um número $O(n/k)$ de cada uma dessas mensagens para que se tenha apenas um processador ativo restante por sub-anel, levando em conta que um processador ao desativar outro herda seus processadores passivos. Assim o número de mensagens da primeira fase é $O(n)$. Na segunda fase temos no máximo $k/2^{\ell-1}$ processadores ativos no passo ℓ, sendo que cada um destes envia $2^{\ell-1}$ mensagens para desativar outros processadores. Conseqüentemente o número de mensagens geradas nesta fase é $\sum_{\ell=1}^{\log k}(k/2^{\ell-1} * 2^{\ell-1}) = O(k \log k)$. Portanto a complexidade

de mensagens é igual a $O(n)$. Se assumirmos que a identificação de um processador possua $O(\log n)$ bits, a complexidade de mensagens em bits do algoritmo é $O(n \log n)$ bits.

No caso de n não ser potência de dois são necessárias algumas modificações no algoritmo. Seja m o menor número maior que n, tal que m é potência de dois. Seja $k = m/2^{\lceil \log \log n \rceil}$. O sub-anel A_i é definido como sendo $\{i, i[k], i[2k], ..., i[(\lceil n/k \rceil - 1)k]\}$. Com n não sendo potência de 2 pode ocorrer de i pertencer a A_j, mas j não pertencer a A_i. Por exemplo, seja $n = 13$, $m = 16$ e $k = 4$. Suponha que os processadores possuam identificadores $1, 2, ..., 13$ e estejam colocados nesta ordem no anel. Então teremos $A_1 = \{1, 5, 9, 13\}$ e $A_9 = \{9, 13, 4, 8\}$. As modificações no algoritmo iniciam ao final da primeira fase, após i desativar todos os processadores em A_i, quando i deverá enviar mensagens *ELEIÇÃO* simultaneamente a todo processador j tal que $i \in A_j$ (o que pode ser calculado já que i conhece n). Somente ao desativar todos estes processadores i pode passar para a segunda fase. Na segunda fase, no passo ℓ, i envia mensagem a todos os processadores com distância múltipla de $k/2^\ell$, menos para aqueles que já foram desativados por i. Além disso i envia mensagem também a todo processador j tal que i está a distância múltipla de $k/2^\ell$ de j. Assim pode-se assegurar que apenas um processador irá declarar-se líder. Com estas modificações, a complexidade de mensagens e de tempo do algoritmo aumentam de um fator constante.

3.2 Eleição em Hipercubos

Uma rede em hipercubo de dimensão d é representada por um grafo com $n = 2^d$ nós rotulados de 0 a $2^d - 1$, tal que existe uma aresta entre quaisquer dois nós se e somente se a representação binária de seus rótulos diferem em um e apenas um bit, sendo que esta aresta é normalmente rotulada com o número do bit que diferencia os nós unidos por ela. Um hipercubo com dimensão d possui $\frac{1}{2}dn$, ou seja $\frac{1}{2}n \log n$, arestas, tendo cada nó grau d.

Se temos disponível os rótulos das arestas para uma rede em hipercubo, podemos dizer que esta rede possui senso de direção. Este tipo de senso de direção será chamado de modelo baseado na dimensão, como o exemplo da figura 3.8 (a). Outro tipo de senso de direção, análogo ao descrito para redes completas, baseado em distância relativa ao ciclo Hamiltoniano, pode ser definido para o hipercubo. Como citado em [FM93], um dos métodos para se construir um ciclo Hamiltoniano em um hipercubo é através do código de Gray. Este código binário refletido é formado por uma seqüência de coordenadas usadas para a determinação do ciclo Hamiltoniano no hipercubo. Por exemplo, para $d = 3$ o código seria (01020102). Iniciando em qualquer nó e seguindo as arestas com rótulos na seqüência do código, um ciclo Hamiltoniano é formado para o hipercubo. Para construirmos um hipercubo com senso de direção baseado nas distâncias relativas a partir de um hipercubo com senso de direção baseado na dimensão, basta determinarmos o código de Gray e rotularmos as arestas baseando-se na distância entre os nós através do ciclo formado usando o código. A figura 3.8(b) mostra um exemplo de hipercubo de dimensão 3 com senso de direção baseado em distância.

O artigo [FM93] apresenta dois algoritmos para eleição em hipercubos com senso de direção e outros dois algoritmos que possibilitam a transição de uma orientação para outra. O algoritmo

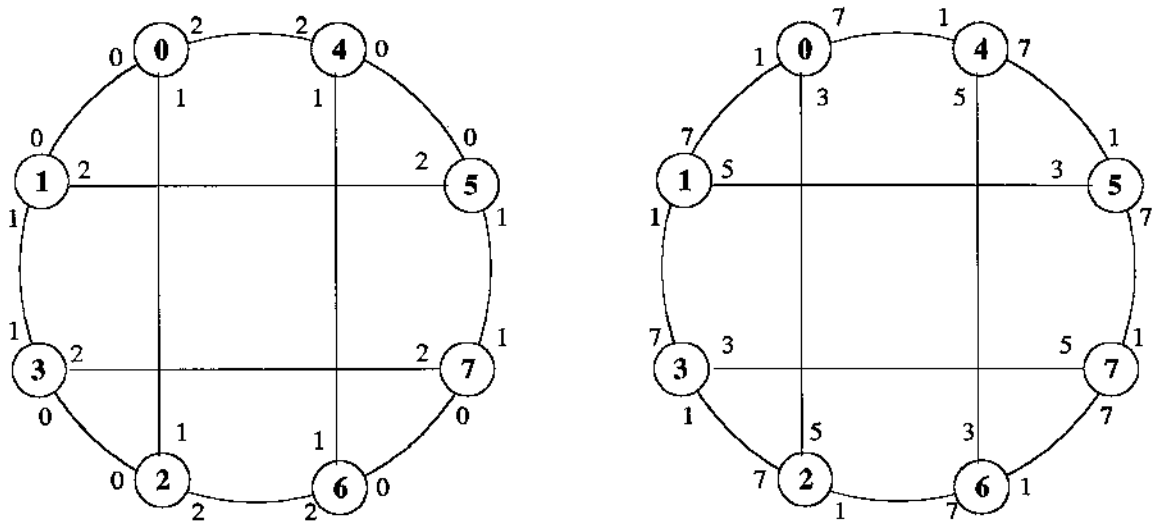


Figura 3.8: Modelos de senso de direção para hipercubos: (a) dimensão (b) distância.

para hipercubo orientado de acordo com as distâncias mostrado em [FM93] não será apresentado aqui pois a eleição é sempre realizada entre apenas 8 processadores, não dando a possibilidade de todos competirem. Caso se tenha um hipercubo orientado de acordo com as distâncias é possível fazer a transformação para a orientação baseada na dimensão e em seguida realizar a eleição. Nesta seção serão apresentados dois algoritmos para eleição em hipercubos assíncronos com senso de direção baseado na dimensão. Estes dois algoritmos, apresentados em [FM93] e [Tel93], foram desenvolvidos no mesmo período mas independentemente e possuem algumas semelhanças que serão descritas em seqüência no texto.

3.2.1 Algoritmo de Flocchini e Mans

Em [FM93] é apresentado um algoritmo para eleição de líder em hipercubo com senso de direção baseado na dimensão, o que significa que os processadores possuem d canais de comunicação com rótulos de 0 a $d-1$. No entanto os processadores não possuem os rótulos binários variando de 0 a $2^d - 1$ normalmente usados em hipercubos.

O algoritmo prossegue em passos. Inicialmente, todos os processadores da rede são *ativos*. Ao final do algoritmo todos os processadores são *passivos* exceto um, que continua *ativo* e declara-se líder. A idéia do algoritmo é fazer com que a disputa ocorra entre os "sub-hipercubos" até se chegar ao hipercubo de dimensão d. Inicialmente (no *passo* 0) todos os n hipercubos de dimensão 0 (compostos de apenas um processador) disputam dois a dois, ou seja cada processador envia uma mensagem *ELEIÇÃO* através do canal com rótulo 0. Os processadores que tiverem identificação maior que a recebida na mensagem vencem suas disputas no primeiro passo, e no passo seguinte devem enviar uma mensagem *ELEIÇÃO* através do canal com rótulo 1, para

desativar o processador ativo de outro sub-hipercubo (agora de dimensão 1). O processador passivo juntamente com o processador que o desativou formam um hipercubo de dimensão 1. Em seguida os $n/2$ processadores ativos dos hipercubos de dimensão 1 disputam dois a dois e formaram $n/4$ hipercubos de dimensão 2. Este procedimento é repetido sucessivamente até que reste um hipercubo de dimensão d e com um único processador ativo.

É fundamental que um processador ao ser desativado mantenha o conhecimento do caminho até o processador que o desativou para que ao receber alguma mensagem de *ELEIÇÃO* no futuro esta mensagem seja retransmitida. Para a construção desse caminho basta ir acrescentando o rótulo dos canais de comunicação que a mensagem *ELEIÇÃO* for percorrendo. De acordo com a formação do hipercubo sabemos que para partir de um processador com rótulo binário $a_3a_2a_1a_0$ e chegar ao processador $\bar{a}_3a_2\bar{a}_1\bar{a}_0$ devemos passar $1+2k$ vezes (sendo k um número natural) pelos canais com rótulos 0, 1 e 3 não importando a ordem, pois sabemos que ao percorrer um canal com rótulo r chegaremos a um processador com rótulo que difere no bit r em relação ao processador na outra extremidade do canal. Ou seja, se temos conhecimento de um caminho entre os processadores i e j dado por <1,2,3>, sabemos que, por exemplo, os caminhos <3,2,1>, <1,2,1,3,1> e <2,1,3> também levam de i a j e de j a i. Na figura 3.9 temos um exemplo de diferentes caminhos entre dois processadores. Em um hipercubo sabemos que a distância máxima entre dois processadores é d, portanto uma forma de representar um caminho entre dois processadores é através de uma seqüência de d bits, sendo que o bit da posição r é um, caso r seja o rótulo de um canal de comunicação que faz parte do caminho entre os processadores. Exemplificando, um caminho <0,2,3> em um hipercubo de dimensão 5 é dado pela seqüência (01101).

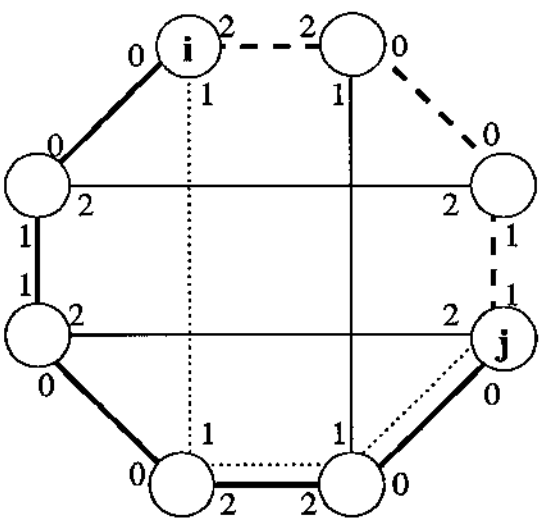


Figura 3.9: Exemplos de caminhos entre os processadores i e j (<1,2,0>, <2,0,1> e <0,1,0,2,0>).

Para o algoritmo, um processador pode ter três valores para seu estado: *ativo, passivo e eleito.* As mensagens do algoritmo são apenas *ELEIÇÃO* e *ELEITO.* Cada processador i inicia a execução do algoritmo, com *estado* = *ativo* e *passo* = 0, enviando uma mensagem

ELEIÇÃO$(i, passo, origem, destino)$, com *origem* e *destino* vazios, através do canal de comunicação com rótulo 0. Os campos *origem* e *destino* na mensagem são seqüências de bits de tamanho $d = \log n$, indicando rótulos de canais de comunicação. Quando um processador torna-se passivo ele armazena a *origem* da mensagem *ELEIÇÃO* acrescentando o número do canal por onde ela foi recebida na variável *líder*, para indicar o caminho até o processador que o desativou. Para fazer essa junção de *origem* com o canal c por onde a mensagem foi recebida devemos substituir o bit c da variável *origem*, com valor b_c, por $1 - b_c$. Sempre que um processador passivo recebe uma mensagem destinada a ele, ou seja com a variável *destino* vazia, ele a envia para o processador que o desativou usando o caminho armazenado na variável *líder*. Na figura 3.10 podemos ver o algoritmo detalhadamente. A função *primeiro* é usada para obter a posição do primeiro bit 1 em uma seqüência de bits. Quando a função $seq \oplus r$ é aplicada o bit r de *seq* com valor b_r é substituído por seu complemento $\bar{b_r}$. Na figura 3.11 temos um exemplo de funcionamento do algoritmo para um hipercubo de dimensão 3.

Para fazer a análise do número de mensagens considere $d(p-1, p)$ como sendo a distância máxima entre um processador i desativado no passo $k-1$ e processador ativo em p que desativou i em $p-1$. Claramente podemos ver que $d(p-1, p) = p$, pois no passo p só foram percorridos canais de comunicação com rótulos de 0 a $p-1$. Seja $F(k)$ o número máximo de canais de comunicação percorridos para que uma mensagem de *ELEIÇÃO* saia de um processador ativo i no passo k e chegue até outro processador ativo, que irá competir com i em k. No passo 0, n processadores ativos enviarão mensagens *ELEIÇÃO* que percorrerão apenas um canal de comunicação ($F(0) = 1$) para chegarem aos concorrentes (processadores ativos em 0). No passo 1 cada uma das $n/2$ mensagens *ELEIÇÃO* deverá percorrer no máximo $F(1) = 1 + d(0,1) = 1 + 1 = 2$ canais para saírem de um processador ativo no passo 1, chegar a um processador que tornou-se passivo no passo 0 e este enviá-la para o processador que o desativou. No próximo passo as $n/2^2$ mensagens deverão percorrer no máximo $F(2) = 1 + d(0,1) + d(1,2) = 4$ (uma mensagem sai do processador ativo no passo 2, chega a um processador que foi desativado no passo 0, este envia uma mensagem para o processador que o desativou no passo 0 que não é mais ativo pois foi desativado no passo 1, assim a mensagem deverá percorrer mais dois canais para chegar ao processador ativo no passo 2). Portanto no passo p uma mensagem *ELEIÇÃO* deverá percorrer, no pior caso,

$$F(p) = 1 + \sum_{k=1}^{p} d(k-1, k) = 1 + \sum_{k=1}^{p} k = 1 + \frac{p.(p+1)}{2}.$$

Assim, o número total de mensagens enviadas pelo algoritmo é no máximo:

$$M = \textit{núm. mens. ELEIÇÃO} + \textit{núm. mens. ELEITO} = \sum_{p=1}^{\log n} \frac{n}{2^{p-1}} F(p-1) \; + \; n - 1$$

$$= \sum_{p=1}^{\log n} \frac{n}{2^{p-1}} \cdot \left(1 + \frac{p.(p-1)}{2}\right) + n - 1 = n \left(\sum_{p=1}^{\log n} \frac{2 + p^2 - p}{2^p} \right) + n - 1$$

```
Algoritmo de Flocchini e Mans para o Processador i
Variáveis:
    estado = ativo;
    passo = 0;
    líder = 0;
    fim = FALSE;
    send ELEIÇÃO(i, passo, 0, 0) to 0;
    while not fim do begin
        if estado = ativo then begin
            receive ELEIÇÃO(j, p, origem, destino) from c;
            if p = passo then begin
                if i > j and p = d − 1 then begin
                    /* d é a dimensão do hipercubo */
                    estado = eleito;
                    for (l = 0; l < d; l++) send ELEITO(i) to l;
                    fim = TRUE;
                end
                if i > j and p < d − 1 then begin
                    passo = passo + 1;
                    send ELEIÇÃO(i, passo, 0, 0) to c + 1;
                end
                if i < j then begin
                    estado = passivo;
                    líder = origem ⊕ c;
                    trata as mensagens pendentes;
                end
            end else if p > passo then begin
                coloca a mensagem em uma fila de mensagens pendentes;
            end
        end
        if estado = passivo then begin
            receive M from c;
            if M = ELEIÇÃO(j, p, origem, destino) then begin
                if destino ≠ 0 then begin
                    l = primeiro(destino);
                    destino = destino ⊕ l;
                    origem = origem ⊕ c;
                    send ELEIÇÃO(j, p, origem, destino) to l;
                end else begin
                    destino = líder;
                    l = primeiro(destino);
                    destino = destino ⊕ l;
                    origem = origem ⊕ c;
                    send ELEIÇÃO(j, p, origem, destino) to l;
                end
            end
            if M = ELEITO(j) then begin
                for (l = 0; l < c; l++) send ELEITO(j) to l;
                fim = TRUE;
            end
        end
    end
```

Figura 3.10: Algoritmo de Flocchini e Mans para hipercubos com senso de direção baseado na dimensão.

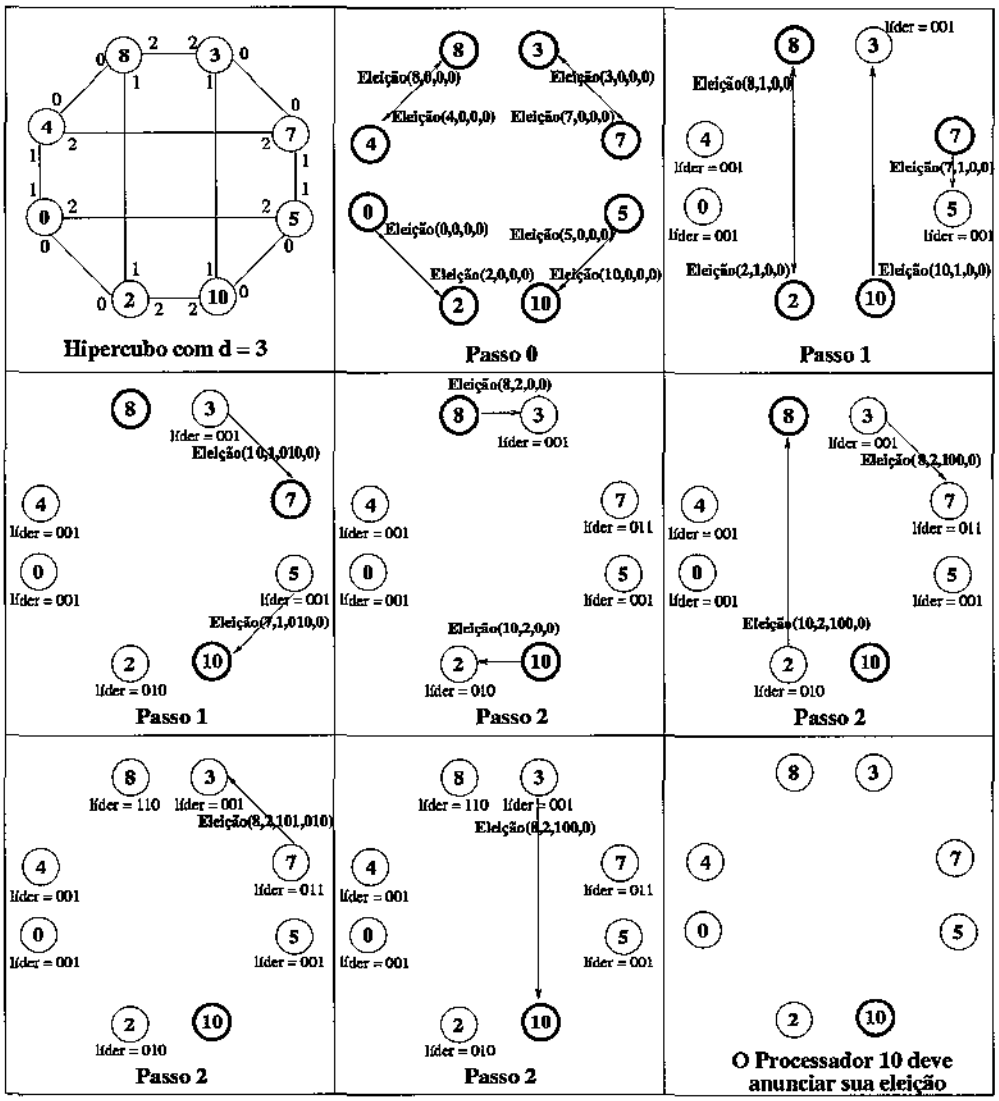


Figura 3.11: Exemplo de funcionamento do algoritmo de Flocchini e Mans.

$$= n\left(\sum_{p=1}^{\log n}\frac{p^2}{2^p} - \sum_{p=1}^{\log n}\frac{p}{2^p} + 2\sum_{p=1}^{\log n}\frac{1}{2^p}\right) + n - 1.$$

Como

$$\sum_{p=1}^{\log n}\frac{p^2}{2^p} = \frac{6n - 4\log n - (\log n)^2 - 6}{n}$$

$$\sum_{p=1}^{\log n}\frac{p}{2^p} = \frac{2n - \log n - 2}{n}$$

$$\sum_{p=1}^{\log n}\frac{1}{2^p} = \frac{n-1}{n},$$

temos que

$$M = 7n - (\log n)^2 - 3\log n - 7.$$

Portanto a complexidade de mensagens é $O(n)$. O tamanho de cada mensagem *ELEIÇÃO* é $\log m + 3\log n$. Considerando que m seja o maior valor para a identificação dos processadores, são necessários $\log m$ bits para a identificação do processador, $\log n$ bits para a variável *passo* e $2\log n$ para a *origem* e o *destino* da mensagem. São gastos $\log n$ passos para que reste apenas um processador ativo, o qual em tempo constante divulga sua eleição. No entanto cada passo pode ter duração de até $\log^2 n$ unidades de tempo para que as mensagens sejam difundidas visto que isto é realizado seqüencialmente. Portanto a complexidade de tempo é $O(\log^3 n)$ unidades de tempo.

3.2.2 Algoritmo de Tel

O algoritmo de Tel [Tel93] também trabalha com a idéia de disputa entre hipercubos de mesma dimensão. No entanto, seu algoritmo é diferente do apresentado na subseção anterior em relação à forma de encontrar o processador ativo em um sub-hipercubo. Ele usa a idéia de se estabelecer conjunto de nós que formam *linhas* e *colunas* no sub-hipercubo. Cada processador pertence a uma única linha e a uma única coluna, sendo que cada linha e coluna possuem apenas um processador em comum.

Considerando um hipercubo de dimensão d e os rótulos binários dos processadores, uma linha de índice $u_{d-1}..u_{\lfloor d/2 \rfloor}$ é definida como sendo o subconjunto de nós $\{u_{d-1}..u_{\lfloor d/2 \rfloor} x_{\lfloor d/2 \rfloor - 1}..x_0\}$ e uma coluna $u_{\lfloor d/2 \rfloor - 1}..u_0$ como sendo o subconjunto de nós $\{x_{d-1}..x_{\lfloor d/2 \rfloor} u_{\lfloor d/2 \rfloor - 1}..u_0\}$. Portanto, temos $2^{\lceil d/2 \rceil}$ linhas e $2^{\lfloor d/2 \rfloor}$ colunas. Exemplificando, em um hipercubo de dimensão 3 existem 4 linhas (linha 00 = {0,1}, linha 01 = {2,3}, linha 10 = {4,5} e a linha 11 = {6,7}) e 2 colunas (coluna 0 = {0,2,4,6} e coluna 1 = {1,3,5,7}). Note que não existe intersecção nem entre as linhas nem entre as colunas (cada nó pertence a uma única linha e única coluna) e que cada linha possui exatamente um nó em comum com cada coluna. Os rótulos binários dados aos processadores do hipercubo não são necessários para o envio de mensagens às linhas e colunas apenas os rótulos das arestas são necessários.

O algoritmo funciona de forma semelhante ao algoritmo de [FM93]. Inicialmente todos os processadores estão ativos e enviam mensagens *ELEIÇÃO*(*id*, *passo*) através do canal de comunicação com rótulo 0, onde *id* é a identificação do processador e *passo* uma variável que indica a dimensão (*passo* + 1) dos hipercubos que estão competindo (no caso *passo* = 0). A eleição ocorre primeiramente entre os sub-hipercubos de dimensão 1, depois entre os de dimensão 2 até que tenhamos um único processador ativo no hipercubo de dimensão d. Um processador ao receber uma mensagem com identificador maior que o seu torna-se passivo, caso contrário ele sabe que venceu a eleição no sub-hipercubo de dimensão = *passo* + 1. Sempre que um processador tomar conhecimento que venceu a competição no passo p ele deve anunciar isso aos processadores de sua linha. Isso é feito enviando uma mensagem *ANÚNCIO* através dos canais com rótulo $\{\lfloor (p+1)/2 \rfloor - 1, .., 0\}$. Um processador ao receber uma mensagem *ANÚNCIO* através de um canal de comunicação c deve retransmiti-la através dos canais com rótulo $\{c-1, .., 0\}$ e

atribuir c a variável que indica o caminho para o processador ativo. O envio das mensagens de anúncio constrói uma árvore geradora que contém todos nós da linha, tendo o processador ativo como raiz da árvore.

Toda mensagem *ELEIÇÃO* parte de um processador ativo de um sub-hipercubo e deve chegar a um processador ativo de um outro sub-hipercubo. Quando um processador passivo recebe uma mensagem *ELEIÇÃO* este deve propagá-la para que a mensagem chegue ao processador ativo. Esta propagação é feita para os processadores que pertencem à coluna do processador passivo que recebeu a mensagem *ELEIÇÃO* diretamente de um processador ativo do outro sub-hipercubo. Como cada coluna possui um processador em comum com cada linha no sub-hipercubo, ao se propagar a mensagem através da coluna, um dos processadores terá conhecimento do processador ativo visto que todos os processadores pertencentes a linha do processador ativo o conhecem. Na figura 3.12 temos o algoritmo de Tel completo. Na figura 3.13 é apresentado um exemplo de funcionamento do algoritmo.

O número de mensagens do algoritmo é dado pelo somatório do número de mensagens de cada passo. Em um passo são enviadas mensagens *ELEIÇÃO* e *ANÚNCIO*. Cada processador ativo envia uma mensagem *ELEIÇÃO* por passo que poderá percorrer todos os processadores que correspondem a uma coluna neste passo. As mensagens *ANÚNCIO* são enviadas pelos processadores que venceram suas competições em suas linhas. Portanto temos que o número total de mensagens do algoritmo é dado por

$$M = \textit{núm. mens. ELEIÇÃO} + \textit{núm. mens. ANÚNCIO} + \textit{núm. mens. ELEITO}$$

$$= \sum_{p=0}^{\log n-1} \frac{n}{2^p}(2^{\lfloor p/2 \rfloor} + 1) + \sum_{p=0}^{\log n-1} \frac{n}{2^{p+1}}(2^{\lfloor (p+1)/2 \rfloor} - 1) + n.$$

Como, caso p seja ímpar, $2^{\lfloor p/2 \rfloor} = \frac{1}{2}2^{p/2}\sqrt{2}$ e $2^{\lceil p/2 \rceil} = 2^{p/2}\sqrt{2}$, no pior caso, temos

$$M = n\left(\sum_{p=0}^{\log n-1} \frac{1}{2^p}(\frac{1}{2}2^{p/2}\sqrt{2} + 1) + \sum_{p=0}^{\log n-1} \frac{1}{2^{p+1}}(2^{(p+1)/2}\sqrt{2} - 1) + 1\right).$$

$$= n\left(1 + \sum_{p=0}^{\log n-1} \frac{2^{p/2}\sqrt{2}}{2^{p+1}} + \sum_{p=0}^{\log n-1} \frac{1}{2^p} + \sum_{p=0}^{\log n-1} \frac{2^{(p+1)/2}\sqrt{2}}{2^{p+1}} - \sum_{p=0}^{\log n-1} \frac{1}{2^{p+1}}\right).$$

$$= n\left(1 + \frac{\sqrt{2}+2}{2} \sum_{p=0}^{\log n-1} \left(\frac{1}{2}\sqrt{2}\right)^p + \frac{1}{2} \sum_{p=0}^{\log n-1} \left(\frac{1}{2}\right)^p\right).$$

Substituindo o $\log n$ por ∞ temos:

$$M < n\left(1 + \frac{\sqrt{2}+2}{2} \sum_{p=0}^{\infty} \left(\frac{1}{2}\sqrt{2}\right)^p + \frac{1}{2} \sum_{p=0}^{\infty} \left(\frac{1}{2}\right)^p\right).$$

$$M < n\left(1 + \frac{\sqrt{2}+2}{2} \cdot \frac{1}{1 - 1/2\sqrt{2}} + 1\right) = 7,82n.$$

```
Algoritmo de Tel para o Processador i
Variáveis:
    estado = ativo;
    passo = 0;
    líder = -1;
    fim = FALSE;
    send ELEIÇÃO(i, passo) to passo;
    while not fim do begin
        if estado = ativo then begin
            receive ELEIÇÃO(j, p) from e;
            if p = passo then begin
                if i > j and p = d − 1 then begin
                    /* d é a dimensão do hipercubo */
                    estado = eleito;
                    for (l = 0; l < d; l++) send ELEITO(i) to l;
                    fim = TRUE;
                end
                if i > j and p < d − 1 then begin
                    for (l = ⌊(passo + 1)/2⌋ − 1; l ≥ 0; l−−) send ANÚNCIO(passo) to l;
                    passo = passo + 1;
                    send ELEIÇÃO(i, passo) to passo;
                end
                if i < j then begin
                    estado = passivo;
                    trata as mensagens pendentes;
                end
            end else if p > passo then begin
                coloca a mensagem em uma fila de mensagens pendentes;
            end
        end
        if estado = passivo then begin
            receive M from e;
            case M of
                ELEIÇÃO(j, p): if passo = p − 1 and líder ≠ −1 then
                                   send ELEIÇÃO(j, p) to líder;
                               if p = e then
                                   for (l = ⌊p/2⌋; l ≤ (p − 1); l++) send ELEIÇÃO(j, p) to l;
                               else for (l = e + 1; l ≤ (p − 1); l++) send ELEIÇÃO(j, p) to l;
                ANÚNCIO(p): líder = e;
                               passo = p;
                               for (l = e − 1; l ≥ 0; l−−) send ANÚNCIO(p) to l;
                ELEITO(j): for (l = 0; l < e; l++) send ELEITO(j) to l;
                líder = e;
                               fim = TRUE;
    end
```

Figura 3.12: Algoritmo de Tel para hipercubos com senso de direção baseado na dimensão.

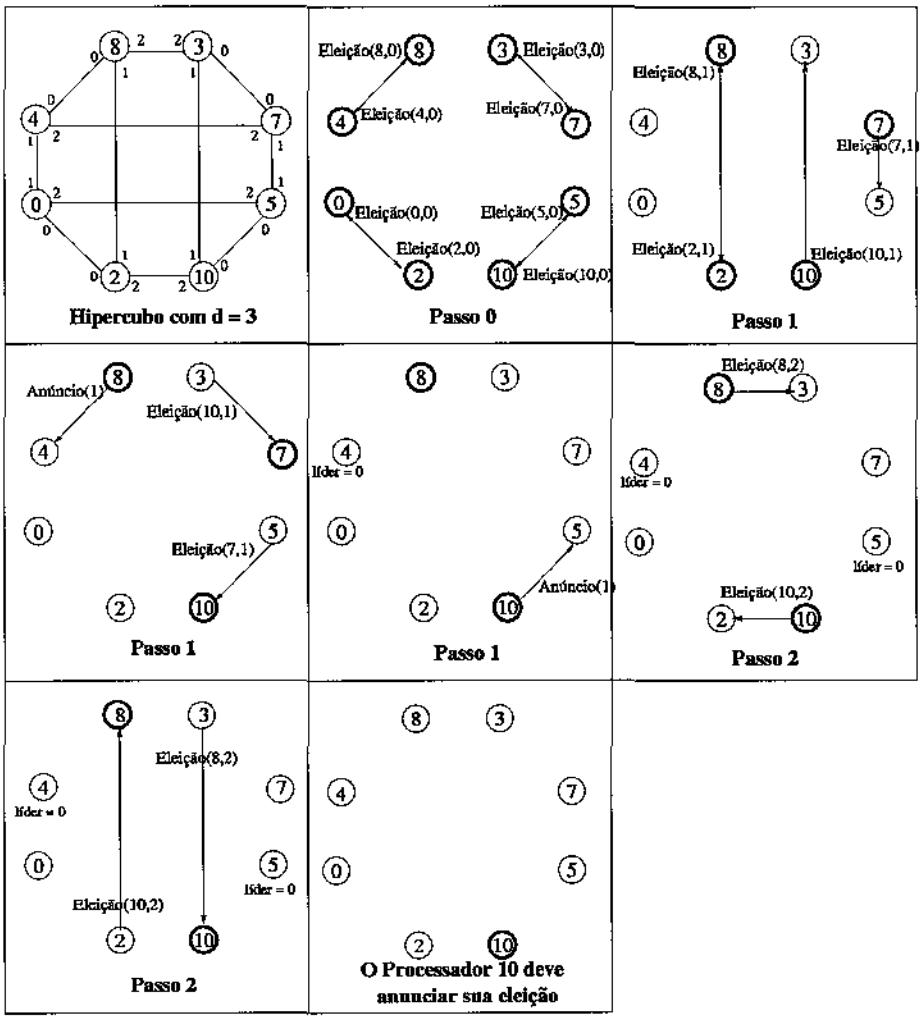


Figura 3.13: Exemplo de funcionamento do algoritmo de Tel.

Logo a complexidade de mensagens é $O(n)$. Considerando que sejam necessários $\log m$ bits para se representar os identificadores dos processadores, o tamanho da maior mensagem é igual a $\log m + \log n$ bits, usados para representar o identificador e a variável *passo* respectivamente. Como o algoritmo executa em passos, sua complexidade de tempo é dada pelo somatório dos tempos de execução de cada passo. Por fazer a difusão da mensagem *ELEIÇÃO* e *ANÚNCIO* em paralelo e não seqüencialmente com em [FM93], cada passo do algoritmo necessita de $O(\log n)$ unidades de tempo para terminar. Assim a complexidade de tempo do algoritmo é igual a $O(\log^2 n)$

Capítulo 4

Algoritmos Auto-estabilizantes

Neste capítulo apresentamos alguns protocolos auto-estabilizantes para eleição de líder. Na primeira seção descrevemos o algoritmo desenvolvido por Dijkstra em [Dij74], que introduziu o termo auto-estabilização em sistemas distribuídos. Nas seções seguintes analisamos dois algoritmos para eleição de líder em anel uniforme e um algoritmo para redes genéricas. Normalmente, para o caso de algoritmo auto-estabilizantes as complexidades estudadas são de tempo e espaço.

4.1 Introdução

O termo auto-estabilizante, usado pela primeira vez na literatura por Dijkstra [Dij74], refere-se a um sistema que inicia em um estado qualquer, possivelmente ilegal, impróprio segundo as regras do sistema e, após um tempo finito, garantidamente alcança uma configuração legítima, ou seja, válida para as definições do sistema. Exemplificando, para o problema de exclusão mútua sabemos que o princípio básico que deve ser obedecido é o de que apenas um processador utilize o recurso compartilhado por vez. Um estado legítimo desse sistema é todo aquele que obedece esse princípio básico.

O artigo de Dijkstra descreve um protocolo auto-estabilizante para exclusão mútua que trabalha em um anel unidirecional não uniforme (nem todos os processadores executam a mesma tarefa). Um processador, denominado processador base, executa um algoritmo diferente do executado pelos outros processadores.

Cada processador no anel possui uma variável inteira indicando seu estado. Para facilitar o entendimento das regras que compõe o protocolo, as seguintes notações são usadas:

- *estado*: refere-se ao próprio estado do nó;
- *estado_viz*: refere-se ao estado de seu vizinho;
- n: é o número total de processadores no anel.

Baseado no seu próprio estado e no estado de seu vizinho, um processador pode ou não possuir o direito de utilizar o recurso compartilhado. Portanto, um estado legítimo do sistema consiste em apenas um processador fazer uso do recurso por vez.

Cada processador deve seguir o seguinte algoritmo:

- para o processador base:

  if ($estado = estado_viz$) **begin**
  $estado = (estado + 1) \bmod n$;
  "processador usa recurso"
  end

- para os outros processadores:

  if ($estado \neq estado_viz$) **begin**
  $estado = estado_viz$;
  "processador usa recurso"
  end

A seguir, na figura 4.1, é mostrado um exemplo do funcionamento do algoritmo. O estado de cada processador é representado pelo número dentro de cada círculo que representam os processadores. O processador base está destacado e os que estão apontados com a seta são os que possuem o recurso no momento. Os estados de A a D são ilegais pois mais de um processador detém o recurso simultaneamente. A partir do estado E o sistema estabiliza-se e apenas um processador usa o recurso por vez.

Provar a correção de protocolos auto-estabilizantes não é uma tarefa fácil [Hua93]. Em [Dij74], Dijkstra não apresenta a prova de correção de seu algoritmo. Ao observar o algoritmo podemos notar que o número de processadores que possuem o direito de utilizar o recurso simultaneamente nunca cresce. Quando um processador executa a mudança de seu estado, ele deixa de possuir o direito e pode fazer com um de seus vizinhos passe a ter esse direito. Devido a assimetria gerada pela presença do processador base, consegue-se que o número de processadores com direito a usar o recurso não permaneça sempre o mesmo (caso seja maior que 1), e sim que vá diminuindo gradativamente, até tornar-se constante e igual a 1.

4.2 Eleição em Anel Uniforme

Nesta seção serão apresentados dois algoritmos auto-estabilizantes em anel uniforme. Os dois protocolos [Hua93, ILS95] trabalham com anel bidirecional de tamanho primo. O tamanho do anel deve ser primo para que possa ser assegurada a auto-estabilização do protocolo.

4.2.1 Algoritmo de Huang

No algoritmo de Huang [Hua93] consideramos um anel de tamanho n, onde n é primo. Cada processador possui um rótulo que inicialmente pode ter qualquer valor arbitrário. Com a execução do protocolo, os rótulos dos processadores tornar-se-ão distintos, variando no domínio

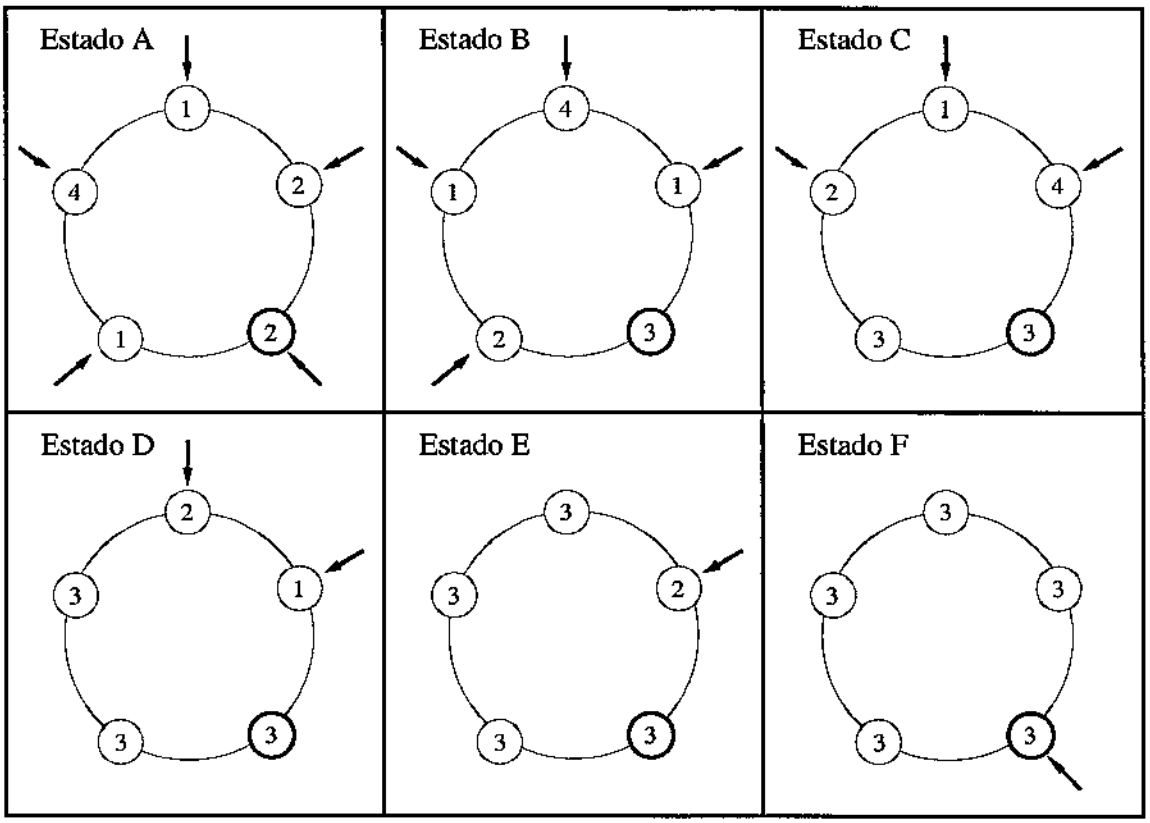


Figura 4.1: Exemplo de funcionamento do algoritmo auto-estabilizante de Dijkstra. O processador destacado é o processador base e as setas indicam os processadores que podem utilizar o recurso.

$\{0, 1, ..., (n-1)\}$, e o protocolo termina. Com isso pode-se, por exemplo, arbitrar que o processador com rótulo 0 será o líder.

O protocolo é apresentado por regras, as quais têm duas partes: a guarda e a mudança de estado. Se a guarda for verdadeira então o processador deve mudar seu estado. O estado do processador é representado por seu rótulo. A mudança de estado é função de seu estado antigo e dos estados de seus vizinhos.

Duas suposições adicionais devem ser feitas. ($A1$) As regras são atômicas, ou seja, um processador não pode avaliar a guarda em um instante e mudar seu estado num instante posterior no qual pode ter ocorrido a mudança de estado de outro processador. ($A2$) Cada processador que pode realizar sua mudança de estado o fará em tempo limitado. Para implementar $A1$ e $A2$ há um Controlador Central[1] que escalona um processador para realizar a mudança de estado.

Sejam a e b rótulos de dois processadores vizinhos no anel. Para a avaliação da condição na guarda é necessária a definição da função $g(a, b)$ para indicar o *gap* de a e b, como sendo:

$$g(a, b) = n, \text{ se } a = b$$

[1] *Central Demon.*

```
(R1): when ((x = y) and (y = n)) begin
          b = (b + 1) mod n
          "mudança de estado"
      end
(R2): when (x < y) begin
          b = (b + 1) mod n
          "mudança de estado"
      end
```

Figura 4.2: Regras do algoritmo de Huang.

```
(R1): when (( x = y ) and ( y = n )) begin
          x = 1
          y = n − 1
          "mudança de estado"
      end
(R2): when ( x < y ) begin
          x = x + 1
          y = y − 1
          "mudança de estado"
      end
```

Figura 4.3: Outra forma de escrever as regras do algoritmo de Huang.

$$= (b - a) \bmod n, \text{ caso contrário.}$$

Pela definição acima, os valores dos *gaps* são inteiros positivos variando entre $\{1, 2, ..., n\}$.

O protocolo consiste de duas regras. As regras, apresentadas a seguir, baseiam-se na notação $a(x)b(y)c$, onde b é o rótulo do processador em consideração; a é o rótulo de seu vizinho da esquerda; c o rótulo de seu vizinho da direita; e x e y não são variáveis dos processadores e sim correspondem a $g(a, b)$ e $g(b, c)$ respectivamente. As regras são apresentadas na figura 4.2, sendo que estas mesmas regras podem ser rescritas como na figura 4.3. A equivalência entre as duas versões das regras pode ser facilmente verificada.

Este protocolo leva o sistema a um estado onde todos os *gaps* são iguais a m $(0 < m < n)$, todos os processadores possuem rótulos distintos e nenhuma das regras é aplicável, ou seja, o protocolo termina. Com o término do protocolo pode-se escolher, por exemplo, o processo com rótulo 0 como líder.

A figura 4.4 mostra um exemplo para $n = 5$ e uma configuração inicial igual a 1(1)2(4)1(1)7(1)3(3), onde os números entre parênteses são os *gaps*. Na figura os rótulos dos processadores são apresentados dentro de cada círculo que os simboliza e os *gaps* são representados pela letra g entre os processadores. As setas indicam os processadores que estão aptos a realizar sua mudança de estado e o processador com o círculo mais espesso será o que executa a mudança de estado em um dado instante no sistema. No estágio G temos a estabilização do sistema, com todos os *gaps* com valor 2 e cada processador com uma identificação única no

intervalo $\{0, 1, ..., n-1\}$.

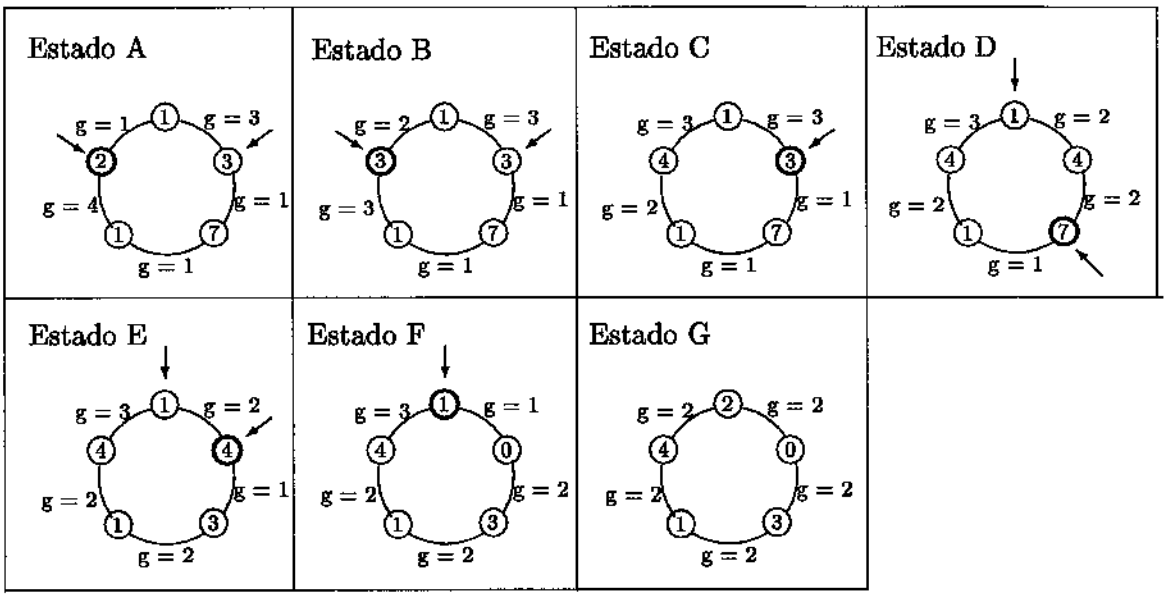


Figura 4.4: Exemplo de funcionamento do algoritmo de Huang.

A correção deste protocolo baseia-se na prova de que o algoritmo termina e quando isso ocorre todos os processadores possuem rótulos distintos. Para a demonstração da terminação do algoritmo, deve-se provar que em algum momento todos os *gaps* possuirão o mesmo valor pois, enquanto houver *gaps* com valores diferentes um dos processadores poderá mudar seu estado. Para isto são definidas três funções: F, G e B. $G \equiv \sum g_i^2$, onde portanto $n \leq G \leq n^3$ pois $1 \leq g \leq n$. B é definida como sendo o número de pares de *gaps* (u, v) tais que $u = v - 1$ e $\forall w : (...u...w...v...) : (w = u) \vee (w = v)$. A notação $(...u...w...v...)$ significa que w é um valor de *gap* entre u e v. Como há $n(n-1)$ pares ordenados de gaps, temos $0 \leq B \leq n(n-1)$. $F \equiv (G, B)$, como G e B são funções limitadas, F também é limitada. É demonstrado em [Hua93] que a cada aplicação de uma das regras R1 ou R2 (mudança de estado do processador) uma das funções G ou B decresce. Com isso, F também decresce e sendo F finita, em algum instante, as regras não poderão mais serem aplicadas. Com a estabilização teremos todos os *gaps* com mesmo valor.

Agora vamos mostrar a demonstração de que todos os rótulos são distintos quando todos os *gaps* possuem o mesmo valor m, $0 < m < n$. Por contradição, suponha que os rótulos não são distintos; portanto dois processadores possuem o mesmo rótulo a e distância k com $0 < k < n$. No teorema 1 do artigo [Hua93] é demonstrado que para quaisquer dois rótulos a e b do anel com distância k entre eles e valores de *gaps* intermediários $g_0, g_1, ..., g_{k-1}$, a relação a seguir é assegurada:

$$\left(\sum_{i=0}^{k-1} g_i\right) \bmod n = (b - a) \bmod n.$$

Seguindo o teorema, $(k * m) \bmod n = (a - a) \bmod n = 0$. Logo, como n é primo, temos $k = 0$ ou $k = n$, o que é contraditório já que $0 < k < n$. Portanto o algoritmo estabilizará quando todos os *gaps* possuírem o mesmo valor e todos os processadores terão rótulos distintos. Assim o processador com rótulo igual a 0 será o líder.

A complexidade de tempo do algoritmo é igual a $O(n)$. Para armazenar cada rótulo dos processadores temos a complexidade de espaço de $O(\log n)$.

4.2.2 Algoritmo de Itkis et al.

Uma outra solução para o problema de eleição de líder auto-estabilizante em anel uniforme é proposta em [ILS95]. Uma das principais características desse algoritmo é o uso de espaço constante de memória, ou seja, complexidade de espaço igual a $O(1)$. O modelo considerado é o de um anel uniforme bidirecional de tamanho primo. Os processadores são idênticos e não possuem identificações. A comunicação é realizada através de memória compartilhada. Existe, como no algoritmo anterior, um escalonador que escolhe os processadores para realizar o movimento. O movimento é uma mudança de estado do processador baseado em seu próprio estado e no estado de seus vizinhos, sendo que este movimento deve ser atômico.

O primeiro campo do estado de um processador é uma variável binária *rótulo*. Um segmento é uma seqüência contígua máxima de processadores com *rótulos* de valores alternados. São chamados de *bordas* os processadores finais de um segmento (aqueles que tem processadores adjacentes com o mesmo rótulo binário), sendo que, obedecendo o sentido horário no anel, temos o processador *início* e o processador *fim* do segmento. Para eliminar segmentos de tamanho unitário, um processador muda seu rótulo caso este seja igual ao de seus dois vizinhos. Na figura 4.5 apresentamos um anel com 11 processadores subdividido em 5 segmentos. O princípio básico de funcionamento para o algoritmo é realizar a união dos segmentos até formar um único segmento no anel sempre com a preocupação de usar espaço constante de memória.

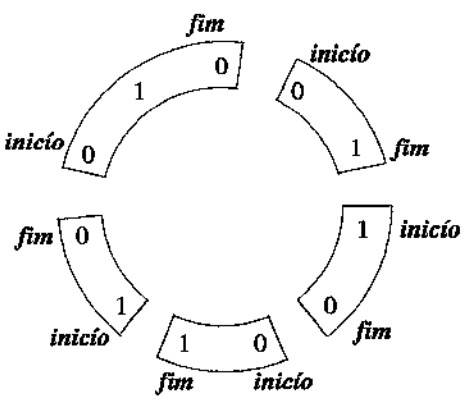


Figura 4.5: Exemplo de anel subdividido em segmentos.

Dois segmentos vizinhos, através de suas bordas (*fim* do segmento da esquerda e *início* do segmento da direita), decidem medir seus comprimentos. O protocolo de medida funciona como segue: as duas bordas se marcam como *MAIS DISTANTE*. O segmento da direita inicia

enviando a ficha medida até encontrar o processador marcado como *MAIS DISTANTE*. Quando o *MAIS DISTANTE* recebe a ficha, ele passa o estado de *MAIS DISTANTE* a seu vizinho na corrente direção. A ficha então muda de direção e retorna ao processador borda. Quando o processador borda recebe a ficha retornado, ele o passa ao processador borda do segmento adjacente, sinalizando que é a vez dele fazer a medida. Este procedimento se repete até que seja encontrado o final de um dos segmentos, ou seja, o processador borda marcado como *mais distante* recebe a ficha.

Ao terminar o protocolo de medida o segmento de comprimento menor (o que chega a seu processador borda primeiro) muda os rótulos binários de seus processadores para juntar-se ao segmento adjacente, com o qual estava realizando a medição. Assim o número de segmentos irá sempre diminuir até que reste apenas um segmento no anel e o processador *início* do segmento será o líder.

A seguir será descrito em detalhes o algoritmo. Quando em um segmento o protocolo de medição é iniciado e monitorado pelo processador *fim* dizemos que temos uma medição da esquerda, analogamente com o processador *início* temos uma medição da direita. Cada processador possui um conjunto de 9 variáveis que são compartilhadas com seus vizinhos da esquerda e direita. As variáveis são:

- 1. *rótulo*: possui valor binário 0,1;
- 2. *vez*: é usada pelos processadores borda para determinar de quem é a vez de fazer a medida (valor binário 0,1);
- 3. *mud_dir* (mudança da direita): possui dois valores *sim*, caso a variável *rótulo* deva ser mudada com a ocorrência de uma ficha de mudança vindo da direita, ou *não*, caso contrário;
- 4. *proc_med_esq* (estado processador na medição da esquerda): pode possuir um de dois valores *mais_distante* ou *normal*, indicando se o processador está marcado como o processador mais distante na execução de uma medição da esquerda;
- 5. *est_med_esq* (estado da medição da esquerda): é usada para fazer o movimento das fichas na medição da esquerda, podendo possuir um de sete valores medição (indica passagem da ficha de medição), volta (para a passagem da ficha de volta da medição), mudança (indica passagem da ficha de mudança que faz a troca dos rótulos de um segmento), limpeza (indica passagem da ficha de limpeza, deve limpar a variável de qualquer possível erro na movimentação das fichas), recebe_volta (para o recebimento da ficha de volta da medição), recebe_mudança (para o recebimento da ficha de mudança) e normal;
- 6. *mud_esq* (mudança da esquerda): semelhante à variável *mud_dir*, agora para o caso de medição da direita;
- 7. *proc_med_dir* (estado processador na medição da direita): semelhante à variável *proc_med_esq*, agora para o caso de medição da direita;

Algoritmo de Itkis et al. para o Processador i
Variáveis:

$vez_i = 0$;
mud_dir_i = não; // mudança da direita
$proc_med_esq_i$ = normal; // estado processador na medição da esquerda
$est_med_esq_i$ = limpeza; // estado da medição da esquerda
mud_esq_i = não; // mudança da esquerda
$proc_med_dir_i$ = normal; // estado processador na medição da direita
$est_med_dir_i$ = limpeza; // estado da medição da direita
$líder_i = 0$;

while TRUE **do begin**
REGRA 1;
REGRA 2;
REGRA 3;
REGRA 4;
REGRA 5;
REGRA 6;
REGRA 7;
REGRA 8;
REGRA 9;
end

Figura 4.6: Algoritmo auto-estabilizante de Itkis et al. para anel uniforme.

- 8. *est_med_dir* (estado da medição da direita): semelhante à variável *est_med_esq*, agora para o caso de medição da direita;
- 9. *líder*: indica se o processador é ou não o líder (valor binário 0,1).

A descrição do algoritmo, apresentada na figura 4.6, é composta de várias regras que são descritas em outras figuras posteriores. As letras sub-escritas e (vizinho esquerdo), d (vizinho direito) e i (processador i) são usadas para indicar a que processador uma variável pertence. A regra 1, apresentada na figura 4.7, é usada apenas para eliminar segmentos de tamanho unitário. Caso haja três processadores consecutivos no anel com mesmo rótulo o do meio deve mudar seu rótulo.

A regra 2, apresentada nas figuras 4.8 e 4.9, refere-se à passagem da ficha de mudança de rótulo e é a segunda a ser executada pois a ficha de mudança é o de maior prioridade e deve ser passado ao próximo vizinho. A ficha de mudança é passado adiante até que seja encontrado um processador com a variável *mud_dir* ou *mud_esq*, dependendo da direção, igual a não, indicando que não deve mudar seu rótulo, ou quando o rótulo do processador for diferente. As regras 2A e 2B são análogas, cada uma referente a uma direção de passagem de ficha (para a esquerda ou direita respectivamente). A única diferença é que em 2B quando a ficha de mudança é eliminado a variável *vez* também é atualizada, para forçar o vizinho direito a limpar suas variáveis quando for selecionado para executar a medição da direita como processador *início*.

Na figura 4.10 é apresentada a regra 3 que consiste na passagem da ficha de limpeza. A ficha

Algoritmo de Itkis et al. para o Processador i
REGRA 1: elimina segmentos de tamanho 1.

```
if rótulo_e = rótulo_i and rótulo_d = rótulo_i then
    rótulo_i = not rótulo_i;
end
```

Figura 4.7: Regra 1.

de limpeza possui a segunda maior prioridade e sua passagem é muito simples. O processador vizinho apenas limpa a variável que indica se ele é o processador *mais distante* e copia a ficha sem que seja necessário pedir para que a ficha seja passado. As partes A e B da regra 3 são simétricas.

Caso um processador P_i esteja esperando para receber uma ficha, mas quem deveria enviá-lo não o fez nem possui mais a ficha, algo de errado aconteceu. O erro é corrigido com P_i inicializando seu estado. Isso ocorre com a espera pelas fichas de mudança e de volta. Na figura 4.11 é apresentada a regra 4, dividida em 4A para a medição da esquerda e 4B para medição da direita.

A regra 5 (figura 4.12) é aplicada quando um processador borda recebe uma ficha de medição. Para um processador *início* P_i, se ele é o *mais distante* em uma medição da esquerda, ele conclui que seu segmento é menor que o segmento da direita, então muda seu rótulo e passa a ficha de mudança para a direita. Caso contrário, P_i envia a ficha de volta de medição. Para um processador *fim* a tarefa é simétrica. A parte A da regra 5 corresponde ao processador *início* e a parte B ao processador *fim*.

Dois processadores borda vizinhos usam a variável *vez* para determinar de quem é a vez de realizar a medição. Considerando um processador borda P_i e seus vizinhos P_e (vizinho esquerdo) e P_d (vizinho direito), se $vez_i + vez_e$ for ímpar e $est_med_dir_i$ = volta então é vez de P_e realizar a medição. Caso contrário a vez é de P_i. A variável *vez* é atualizada quando o processador borda recebe a ficha de volta. Um processador *início* em sua vez sempre tentará executar um novo protocolo de medição (para caso alguma ficha se perca) e sempre envia uma ficha de limpeza. Na figura 4.13 é apresentada a regra 6 que realiza a criação das fichas de medição e limpeza, descrita acima. A primeira parte da regra (6A) é executada pelo processador *início* e a segunda parte (6B) pelo processador *fim*. As duas partes não são simétricas, pois o processador *início* sempre começa o protocolo de medida.

A figura 4.14 apresenta a regra 7 que controla a passagem da ficha de volta da medição que irá eliminar qualquer ficha de medição encontrado. Antes de passar a ficha de volta de medição, o processador copia o valor do estado do processador vizinho na medição (*mais_distante* ou *normal*). Isso fará com que a marca de mais distante seja passada adiante. Quando um processador recebe a ficha de volta da medição, ele troca o valor de sua variável *vez* para que a vez de medir seja passada ao outro segmento. As duas partes da regra 7 são simétricas.

Algoritmo de Itkis et al. para o Processador i
REGRA 2A: passagem da ficha mudança.

```
if (est_med_dir_d = mudança) and (est_med_dir_i ≠ mudança) and (mud_esq_i = sim)
and (rótulo_i = rótulo_d) then begin
    proc_med_dir_i = normal;
    est_med_dir = recebe_mudança; // pede para receber a ficha de mudança
end
if ((est_med_dir_i = recebe_mudança) and (est_med_dir_d = normal)) and (mud_esq_i = sim)
and (rótulo_i = rótulo_d) then begin
    // recebe a ficha de mudança e troca seu rótulo
    rótulo_i = not rótulo_i;
    vez_i = 0;
    mud_dir_i = não;
    proc_med_esq_i = normal;
    est_med_esq_i = limpeza;
    mud_esq_i = não;
    proc_med_dir_i = normal;
    est_med_dir_i = mudança;
end
else if (est_med_dir_i = recebe_mudança) and (est_med_dir_d = normal) then begin
    // elimina a ficha de mudança
    mud_esq_i = não;
    proc_med_dir_i = normal;
    est_med_dir_i = limpeza;
end
if (est_med_dir_i = mudança) then begin
    if (est_med_dir_e = recebe_mudança) then begin
        // passa a ficha de mudança ao próximo
        mud_esq_i = não;
        proc_med_dir_i = normal;
        est_med_dir_i = normal;
    end
    else if (mud_esq_e = não) or (rótulo_e ≠ rótulo_i) then begin
        // elimina a ficha de mudança
        mud_esq_i = não;
        proc_med_dir_i = normal;
        est_med_dir_i = limpeza;
    end
end
```

Figura 4.8: Regra 2A.

Algoritmo de Itkis et al. para o Processador i
REGRA 2B: passagem da ficha mudança.

```
if (est_med_esq_e = mudança) and (est_med_esq_i ≠ mudança) and (mud_dir_i = sim)
and (rótulo_e = rótulo_i) then begin
      proc_med_esq_i = normal;
      est_med_esq_i = recebe_mudança; // pede para receber a ficha de mudança
end
if ((est_med_esq_i = recebe_mudança) and (est_med_esq_e = normal)) and (mud_dir_i = sim)
and (rótulo_i = rótulo_e) then begin
      // recebe a ficha de mudança e troca seu rótulo
      rótulo_i = not rótulo_i;
      vez_i = 0;
      mud_dir_i = não;
      proc_med_esq_i = normal;
      est_med_esq_i = mudança;
      mud_esq_i = não;
      proc_med_dir_i = normal;
      est_med_dir_i = limpeza;
end
else if (est_med_esq_i = recebe_mudança) and (est_med_esq_e = normal) then begin
      // elimina a ficha de mudança e copia a variável vez
      vez_i = vez_d;
      mud_dir_i = não;
      proc_med_esq_i = normal;
      est_med_esq_i = limpeza;
end
if (est_med_esq_i = mudança) then begin
      if (est_med_esq_d = recebe_mudança) then begin
            // passa a ficha de mudança ao próximo
            mud_dir_i = não;
            proc_med_esq_i = normal;
            est_med_esq_i = normal;
      end
      else if (mud_dir_d = não) or (rótulo_d ≠ rótulo_i) then begin
            // elimina a ficha de mudança
            if (est_med_dir_d = volta) then begin
                  vez_i = vez_d; // copia a variável vez
                  mud_dir_i = não;
                  proc_med_esq_i = normal;
                  est_med_esq_i = limpeza;
            end
            else begin
                  vez_i = not vez_d; // atualiza a variável vez
                  mud_dir_i = não;
                  proc_med_esq_i = normal;
                  est_med_esq_i = limpeza;
            end
      end
end
```

Figura 4.9: Regra 2B.

Algoritmo de Itkis et al. para o Processador i
REGRA 3: passagem da ficha limpeza.

REGRA 3A:
if ($rótulo_i \neq rótulo_e$) **and** ($est_med_dir_e$ = limpeza) **then begin**
$est_med_dir_i$ = limpeza;
$proc_med_dir_i$ = normal;
end
REGRA 3B:
if ($rótulo_i \neq rótulo_d$) **and** ($est_med_esq_d$ = limpeza) **then begin**
$est_med_esq_i$ = limpeza;
$proc_med_esq_i$ = normal;
end

Figura 4.10: Regra 3.

Algoritmo de Itkis et al. para o Processador i
REGRA 4: correção de erro.

REGRA 4A:
if ($est_med_dir_i$ = recebe_mudança) **and** (($est_med_dir_d \neq$ mudança) **and** ($est_med_dir_d \neq$ normal))
then begin
mud_esq_i = não;
$proc_med_dir_i$ = normal;
$est_med_dir_i$ = limpeza;
end
if ($est_med_dir_i$ = recebe_volta) **and** (($est_med_dir_d \neq$ volta) **and** ($est_med_dir_d \neq$ normal))
then begin
mud_esq_i = não;
$proc_med_dir_i$ = normal;
$est_med_dir_i$ = limpeza;
end
REGRA 4B:
if ($est_med_esq_i$ = recebe_mudança) **and** (($est_med_esq_e \neq$ mudança) **and** ($est_med_esq_e \neq$ normal))
then begin
mud_dir_i = não;
$proc_med_esq_i$ = normal;
$est_med_esq_i$ = limpeza;
end
if ($est_med_esq_i$ = recebe_volta) **and** (($est_med_esq_e \neq$ volta) **and** ($est_med_esq_e \neq$ normal))
then begin
mud_dir_i = não;
$proc_med_esq_i$ = normal;
$est_med_esq_i$ = limpeza;
end

Figura 4.11: Regra 4.

Algoritmo de Itkis et al. para o Processador i
REGRA 5: criação das fichas mudança e volta.

REGRA 5A:
if $((rótulo_e = rótulo_i)$ **and** $(rótulo_i \neq rótulo_d))$ **and**
$(est_med_esq_i =$ medição$)$ **then begin**
if $(proc_med_esq_e =$ mais_distante$)$ **then begin**
// muda seu rótulo e inicia a passagem da ficha de mudança
$rótulo_i =$ **not** $rótulo_i$;
$vez_i = 0$;
$mud_dir_i =$ não;
$proc_med_esq_i =$ normal;
$est_med_esq_i =$ mudança;
$mud_esq_i =$ não;
$proc_med_dir_i =$ normal
$est_med_dir_i =$ limpeza;
end
else
// inicia a passagem da ficha de volta da medição
$proc_med_esq_i =$ normal;
$est_med_esq_i =$ volta;
end
end
REGRA 5B:
if $((rótulo_d = rótulo_i)$ **and** $(rótulo_i \neq rótulo_e))$ **and**
$(est_med_dir_i =$ medição$)$ **then begin**
if $(proc_med_dir_d =$ mais_distante$)$ **then begin**
// muda seu rótulo e inicia a passagem da ficha de mudança
$rótulo_i =$ **not** $rótulo_i$;
$vez_i = 0$;
$mud_dir_i =$ não;
$proc_med_esq_i =$ normal;
$est_med_esq_i =$ limpeza;
$mud_esq_i =$ não;
$proc_med_dir_i =$ normal;
$est_med_dir_i =$ mudança;
end
else
// inicia a passagem da ficha de volta da medição
$proc_med_dir_i =$ normal;
$est_med_dir_i =$ volta;
end
end

Figura 4.12: Regra 5.

BIBLIOTECA CENTRAL

Algoritmo de Itkis et al. para o Processador i
REGRA 6: criação das fichas medição e limpeza.

```
REGRA 6A:
if (rótulo_e = rótulo_i) and (rótulo_i ≠ rótulo_d) then begin
    if (est_med_dir_i = volta) and (vez_e + vez_i = 1) then begin
        exit; // vez do vizinho esquerdo
    end
    if (vez_e + vez_i = 1) then begin
        // envia ficha de limpeza para reinicia o protocolo de medição
        est_med_dir_i = limpeza;
        vez_i = rótulo_e ; // mudar vez_i tal que (vez_i + vez_e) seja par
    end
    if ((est_med_dir_i = limpeza) and (est_med_dir_d = limpeza) and (est_med_esq_e = limpeza)) or
    ((est_med_esq_e = volta) and (est_med_dir_i = volta)) then begin
        // inicia protocolo de medição
        mud_esq_i = sim;
        proc_med_dir_i = mais_distante;
        est_med_dir_i = medição;
    end
    else if (est_med_dir_i ≠ medição) then begin
        // aconteceu algum erro, é necessário limpar
        mud_esq_i = não;
        proc_med_dir_i = normal;
        est_med_dir_i = limpeza;
    end
end

REGRA 6B:
if (rótulo_d = rótulo_i) and (rótulo_i ≠ rótulo_e) then begin
    if not ((est_med_dir_d = volta) and (vez_e + vez_d = 1)) then begin
        if (est_med_dir_d = limpeza) then
            est_med_esq_i = limpeza; // faz limpeza
        end
        exit; // a vez é do outro processador
    end
    if((est_med_esq_e = limpeza) and (est_med_esq_i = limpeza)) or (est_med_esq_i = volta) then begin
        // inicia protocolo de medição
        mud_dir_i = sim;
        proc_med_esq_i = mais_distante;
        est_med_esq_i = medição;
    end
    else if (est_med_esq_i ≠ medição) then begin
        // aconteceu algum erro, é necessário limpar
        mud_dir_i = não;
        proc_med_esq_i = normal;
        est_med_esq_i = limpeza;
    end
end
```

Figura 4.13: Regra 6.

Algoritmo de Itkis et al. para o Processador i
REGRA 7: passagem da ficha volta.

REGRA 7A:
if ($rótulo_i \neq rótulo_d$) **then begin**
  if ($est_med_dir_d$ = volta) **then**
    $est_med_dir_i$ = recebe_volta; // pede para receber a ficha de volta de medição
  end
  if ($est_med_dir_i$ = recebe_volta) **and** ($est_med_dir_d$ = normal) **then begin**
    // recebe a ficha de volta de medição e troca o valor da variável *vez*
    vez_i = **not** vez_i;
    $est_med_dir_i$ = volta;
  end
  if ($est_med_esq_i$ = volta) **and** ($est_med_esq_d$ = recebe_volta) **then begin**
    // copia o estado do processador vizinho e passa a ficha de volta de medição
    $proc_med_esq_i$ = $proc_med_esq_d$;
    $est_med_esq_i$ = normal;
  end
end

REGRA 7B:
if ($rótulo_i \neq rótulo_e$) **then begin**
  if ($est_med_esq_e$ = volta) **then**
    $est_med_esq_i$ = recebe_volta; // pede para receber a ficha de volta de medição
  end
  if ($est_med_esq_i$ = recebe_volta) **and** ($est_med_esq_e$ = normal) **then begin**
    // recebe a ficha de volta de medição e troca o valor da variável *vez*
    vez_i = **not** vez_i;
    $est_med_esq_i$ = volta;
  end
  if ($est_med_dir_i$ = volta) **and** ($est_med_dir_e$ = recebe_volta) **then begin**
    // copia o estado do processador vizinho e passa a ficha de volta de medição
    $proc_med_dir_i$ = $proc_med_dir_e$;
    $est_med_dir_i$ = normal;
  end
end

Figura 4.14: Regra 7.

Algoritmo de Itkis et al. para o Processador i
REGRA 8: passagem da ficha medição.

REGRA 8A:
if ($rótulo_i \neq rótulo_e$) **and** ($est_med_dir_e$ = medição) **then begin**
  if ($est_med_dir_i \neq$ limpeza) **or** ($est_med_dir_d$ = limpeza) **or** ($rótulo_i = rótulo_d$) **then begin**
    mud_esq_i = sim;
    $est_med_dir_i$ = medição;
  end
end
REGRA 8B:
if ($rótulo_i \neq rótulo_d$) **and** ($est_med_esq_d$ = medição) **then begin**
  if ($est_med_esq_i \neq$ limpeza) **or** ($est_med_esq_e$ = limpeza) **or** ($rótulo_i = rótulo_e$) **then begin**
    mud_dir_i = sim;
    $est_med_esq_i$ = medição;
  end
end

Figura 4.15: Regra 8.

Algoritmo de Itkis et al. para o Processador i
REGRA 9: eleição do líder.

if ($est_med_dir_i$ = medição) **then begin**
  if ($rótulo_i = rótulo_e$) **and** ($rótulo_i \neq rótulo_d$) **then begin**
    $líder_i = 1$;
  else
    $líder_i = 0$;
  end
end

Figura 4.16: Regra 9.

A regra 8 apresentada na figura 4.15 apenas realiza a passagem da ficha de medição. Essa passagem é semelhante à da ficha de limpeza. Não é necessário pedir para que a ficha seja passado, o processador apenas copia a ficha. As duas partes da regra também são simétricas.

Na regra 9 (figura 4.16), caso um processador tenha a ficha de medição e seja o processador *início* do segmento ele se marca como líder, caso contrário ele passa sua variável *líder* para o valor 0.

Na figura 4.17 pode ser visto um exemplo de como funciona a junção dos segmentos para o protocolo de Itkis et al. Os processadores estão representados por seus rótulos binários e o processador que possui a ficha apresenta um círculo em torno de seu rótulo. O círculo contínuo menos espesso indica a ficha de medição, o círculo tracejado representa a presença da ficha de volta de medição e o círculo mais espesso é a ficha de mudança que vem mudando os rótulos dos processadores realizando a junção dos segmentos.

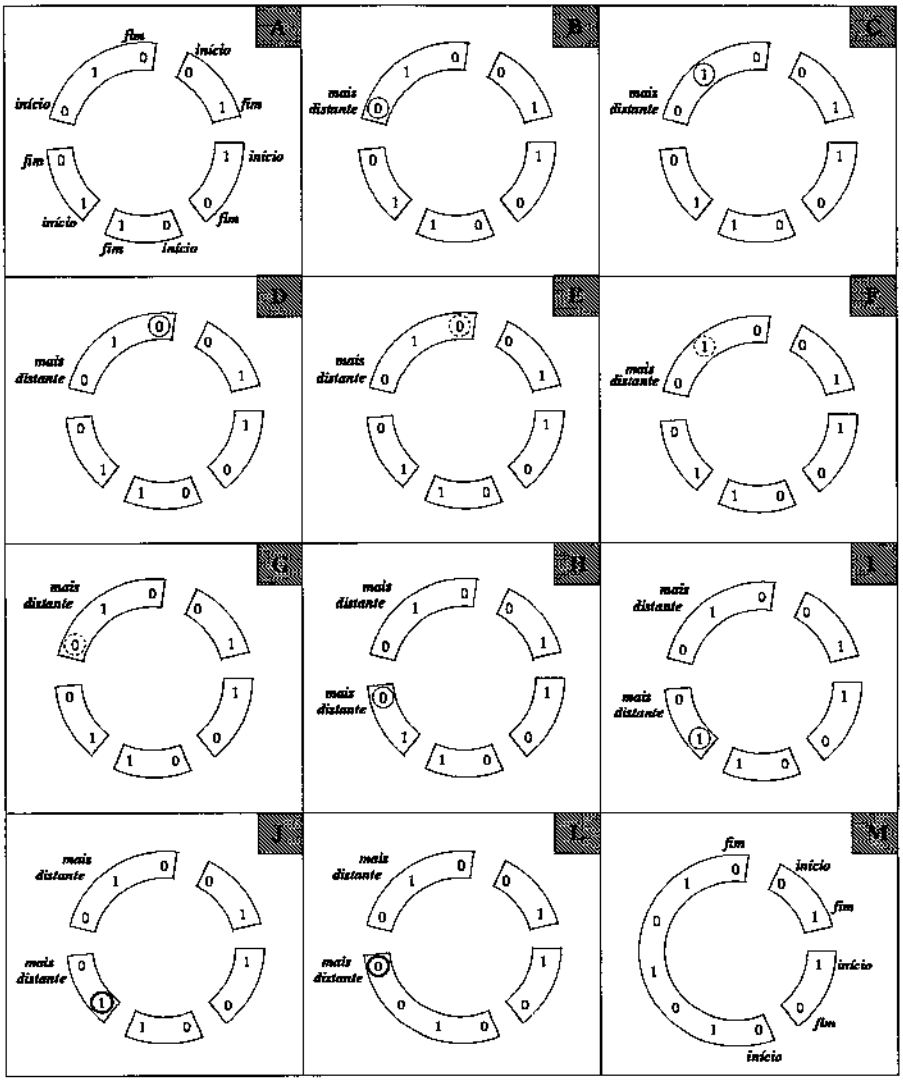


Figura 4.17: Exemplo de funcionamento do algoritmo de Itkis et al.

Este algoritmo está livre de *deadlock* visto que a cada configuração do sistema algum dos processadores estará apto a mudar seu estado, pois o processador *início* sempre tentará iniciar uma nova medição e com a mudança de estado de um processador, um de seus vizinhos estará apto a também mudar seu estado. Com a execução do algoritmo, o número de segmentos sempre diminuirá, visto que com a medição o processador de tamanho menor irá juntar-se ao maior. Devemos lembrar que o número de processadores no anel é primo, portanto haverá sempre segmentos de tamanhos diferentes. Assim, o protocolo se auto-estabilizará quando possuir apenas um segmento e o processador *início* será o líder.

Em [ILS95] é demonstrado que o protocolo estabiliza em tempo finito, utilizando $O(n^2)$ passos. Considere dois segmentos adjacentes A e B com comprimentos x e y respectivamente, onde $x < y$. Para o segmento A se unir a B é necessário que cada processador de A e B sejam

ativados (mudem seus estados) $O(x)$ vezes, para que se encontre o final do menor segmento e se realize a mudança dos rótulos. Logo, para que todos os processadores do anel formem um único segmento, são necessários $O(n^2)$ passos, com complexidade de espaço igual a $O(1)$.

4.3 Eleição em Redes Genéricas

Várias técnicas podem ser empregadas para o desenvolvimento de algoritmos para eleição de líder em redes genéricas (uma topologia qualquer) [Lyn96]. Algumas dessas técnicas são: a difusão de mensagens através da rede através de uma busca em profundidade, a difusão de mensagens através de uma árvore geradora da rede, usar um sincronizador para simular um algoritmo síncrono, usar um mecanismo de detecção do estado global para determinar a terminação de um algoritmo assíncrono.

Em [DIM97] é apresentado um algoritmo auto-estabilizante para redes genéricas, anônimas e uniformes. O modelo considerado pelo algoritmo de Dolev *et al.* consiste de n processadores denotados por P_1, P_2,..., P_n para facilitar a notação, pois em redes anônimas os processadores não possuem identificadores. A comunicação é realizada através de registradores com escritor único e múltiplos leitores. Cada processador é considerado uma CPU que executa programas compostos de passos atômicos. Um passo atômico consiste de uma computação interna seguida por uma ação de terminação. Uma ação de terminação pode ser uma operação de leitura, escrita ou "lançamento de uma moeda" (escolher um valor entre dois com a mesma probabilidade). Uma *configuração* c do sistema é representada por um vetor dos estados de todos os seus processadores.

A ativação dos processadores é gerenciada por um escalonador. O escalonador escolhe qual o próximo processador a ser ativado, usando a informação sobre a execução até o momento. Uma *execução* do sistema é uma seqüência finita ou infinita de configurações $E = (c_1, c_2, ...)$, tal que para $i = 1, 2, ...$, a configuração c_{i+1} é alcançada a partir de c_i com um dos processadores executando um único passo atômico.

Para o cálculo da complexidade de tempo deste algoritmo é definida uma unidade de tempo de execução chamada *round*. Dada uma execução E, definimos o primeiro *round* de E como sendo o prefixo minimal de E, E', que contém passos atômicos de todos os processadores do sistema. Assim a complexidade de tempo de uma execução E é o número de *rounds* de E. Com essa definição o tempo para completar um *round* não é determinado e depende do escalonador.

Neste protocolo cada configuração do sistema possui um grafo dirigido chamado GPF (grafo de relação pai-filho). Uma configuração segura do sistema para este protocolo é uma configuração cujo grafo não dirigido baseado em GPF é uma árvore enraizada que contém todos os processadores e, para a qual durante qualquer execução que inicie nesta configuração a árvore não muda. Ao obter-se uma configuração segura, o processador raiz da árvore é eleito líder. O GPF é codificado através da variável pai_i que cada processador possui. Cada processador pode ser uma raiz ou ter um pai, que é um de seus vizinhos. Se P_i é pai de P_j em uma configuração c, então existe uma aresta dirigida de P_j para P_i. Cada árvore em GPF é identificada por um identificador de árvore a_{id} e a raiz da árvore é o único processador a modificar o valor da identificação da árvore estendendo com um bit aleatório. Esse novo valor para a_{id} é propagado

para o restante da árvore pois os processadores que não são raízes copiam repetidamente o valor de a_{id} de seus pais.

O protocolo consiste de duas fases, chamadas *eliminação de ciclos* e *fusão de árvores*. Durante a eliminação de ciclos é obtida uma floresta que é reduzida a uma única árvore na fase de fusão. Pela natureza do modelo do protocolo não há como um processador detectar localmente se todos os ciclos do grafo foram eliminados. Portanto, pode ser que a fase de fusão das árvores comece antes mesmo de se eliminar todos os ciclos do grafo.

Para eliminar ciclos um processador quebra a ligação com o pai e torna-se raiz sempre que toma conhecimento que possui identificação de árvore maior que seus vizinhos ou quando aumenta a distância entre ele e a raiz de sua árvore, calculada adicionando um à distância de seu pai até a raiz.

Para reduzir o número de árvores a *um* deve ser assegurado que em algum momento haverá apenas um *aid* no sistema e uma única árvore com esse *aid*. Um processador raiz descobre que há outras árvores com o mesmo *aid* colorindo sua árvore repetidas vezes. Cada coloração inicia-se com a raiz escolhendo aleatoriamente uma nova cor entre seis valores (0 a 7 menos a cor atual e a cor anterior). A seguir a raiz espera até que todos os seus filhos confirmem que suas sub-árvores foram coloridas com a nova cor. A detecção da existência de uma outra árvore com o mesmo a_{id} é realizada através da observação da cor de todos os vizinhos. Caso um processador encontre um vizinho com cor diferente da cor anterior da árvore e da cor atual, ele toma conhecimento de que este vizinho pertence a outra árvore. Esta informação é retornada à raiz. Ao receber esta informação, a raiz estende *aid* adicionando ao seu final um bit escolhido aleatoriamente.

O algoritmo, mostrado na figura 4.18, é formado por um laço infinito executado por todos os processadores. No início do laço cada processador lê o registro de variáveis(r_j) de todos seus vizinhos para um conjunto de variáveis locais. As variáveis que compõem o registro são:

- aid_i: identificação da árvore a que P_i pertence;
- dis_i: indica a distância de P_i à raiz da árvore a que ele pertence;
- pai_i: para $dis_i \neq 0$, pai_i deve conter o índice do canal que liga P_i a seu pai (um de seus vizinhos);
- cor_i: é um inteiro variando de 0 a 7 usado para colorir a árvore;
- $ciente_i$: usada pelo protocolo de sincronização global;
- na_i: esta variável lógica terá valor *verdadeiro* sempre que um processador tomar conhecimento de que existe algum processador com cor diferente tanto da nova cor escolhida pela raiz quanto da cor corrente, ou seja, existe uma nova árvore no sistema.

A variável *cor_anterior*, usada pela raiz, contém a cor anterior com a qual a árvore foi colorida. O grau máximo de um nó do grafo é denotado por Δ. A função *Estende_aid* escolhe um bit aleatoriamente e o concatena ao final de *aid*. A função $filho(j)$ quando executada pelo

processador P_i retorna *verdadeiro* se P_j é filho de P_i, baseada no valor de $pai_i[j]$. A função *Escolhe_cor* escolhe aleatoriamente uma cor dentre um conjunto de seis valores (0 a 7 menos a cor atual da árvore e a cor anterior).

Considere dois valores aid_1 e aid_2 para a variável *aid*. A relação $aid_1 > aid_2$ é *verdadeira* quando aid_2 é prefixo de aid_1 ou se o primeiro bit de aid'_1 é 1, sendo aid'_1 e aid'_2 derivados de aid_1 e aid_2 removendo o prefixo maximal comum. A relação $((aid_1, dis_1) \succ (aid_2, dis_2))$ é válida se $aid_1 > aid_2$ ou se $aid_1 = aid_2$ e $dis_1 < dis_2$.

Na figura 4.19 pode ser visto um exemplo do funcionamento do algoritmo de Dolev *et al.*. No estágio A da figura podemos ver as ligações entre os processadores. As variáveis *aid*, *dis*, *cor*, *ciente* e *na* de cada processador estão apresentadas entre parênteses nessa ordem. A variável *pai* é representada pela seta entre os processadores. A cada estágio é escolhido um processador, apresentado de forma destacada, para executar o algoritmo. Os processadores iniciam com valores quaisquer para as variáveis *aid*, *dis*, *pai* e *cor*. O processador com maior *aid* em sua vizinhança torna-se raiz. Nos estágios E e F os processadores escolhidos para executar têm duas opções para seus pais (dois processadores com mesmo *aid* máximo e *dis* mínimo) e podem escolher qualquer um dos dois. No estágio G temos duas árvores com mesmo *aid* e em H o processador raiz de uma das árvores estende seu *aid* para que seja iniciada a fusão das árvores. Ao final do estágio N temos uma única árvore.

Como o protocolo faz uso da abstração de variáveis compartilhadas, ele utiliza dois protocolos para sincronização um local e outro global. O protocolo de sincronização local é projetado para um sistema com dois processadores. O protocolo de sincronização global é uma extensão do protocolo local para um sistema com estrutura de árvore. A fase de fusão de ciclos utiliza o protocolo de sincronização global.

Após um certo tempo de execução, o algoritmo de eleição leva à configuração de uma floresta, sendo que a configuração do sistema permanece uma floresta pelo resto da execução. Esta floresta é convertida em uma única árvore que não é mais alterada; o processador raiz é declarado líder. O tempo gasto para a eleição é $O(\Delta\, D\, \log n)$ *rounds*, onde n é o número de processadores na rede, Δ é o número máximo de vizinhos de um processador, D é o diâmetro da rede, ou seja, a distância máxima entre dois processadores. O algoritmo possui uma complexidade de espaço igual a $O(\log n)$ para armazenar o estado de cada processador.

Algoritmo de Dolev et al.

```
while (TRUE) do
begin
    for (j = 1) to Δ do
        (aid_i[j], dis_i[j], pai_i[j], cor_i[j], ciente_i[j], na_i[j]) = read(r_j);
    max_aid = max{aid_i[j]};
    min_dis = min{aid_i[j] | aid_i[j] = max_aid}+1;
    if (aid_i[pai_i], dis_i[pai_i]) = (max_aid, min_dis − 1) then
        (F, COR, CI, NA) = (pai_i, cor_i, ciente_i, na_i);
    else begin
        F = { primeiro j | aid_i[j] = max_aid and dis_i[j] = min_dis − 1};
        (COR, CI, NA) = (cor_i[F], false, false);
    end
    if (aid_i, dis_i) ≻ (max_aid, min_dis) then
        write (dis_i, pai_i) = (0, nil);
    else
        write (aid_i, dis_i, pai_i, cor_i, ciente_i, na_i) = (max_aid, min_dis, F, COR, CI, NA);
    if (∀j| (aid_i[j] = aid_i) and (|dis_i − dis_i[j]| ≤ 1) then
        Fusão_de_Árvores();
end

Procedure Fusão_de_Árvores
    if (dis_i = 0) and {∀j|filho(j) ⇒ ((cor_i[j] = cor_i) and ciente_i[j])} then
    begin
        if { ∃j| (na_i[j] = TRUE) or (cor_i[j] ≠ cor_i)} then
            write (aid_i, ciente_i) = (Estende_aid(aid_i), false);
        write cor_i = Escolhe_cor(cor_anterior, cor_i);
    end
    else if (dis_i ≠ 0) then
        if (cor_i ≠ cor_i[pai_i]) then
        begin
            if { ∃j| (cor_i[j] ≠ cor_i and cor_i[j] ≠ cor_i[pai_i]) } then
                write na_i = TRUE;
            else
                write na_i = false;
            write (cor_i, ciente_i) = (cor_i[pai_i], false);
        end
        else
            if not ciente_i and {∀j|filho(j) ⇒ ((cor_i[j] = cor_i) and ciente_i[j])} then
            begin
                if { ∃j| filho(j) and na_i[j] = TRUE} then
                    na_i = TRUE;
                write ciente_i = TRUE;
            end
end
```

Figura 4.18: Algoritmo auto-estabilizante para redes genéricas.

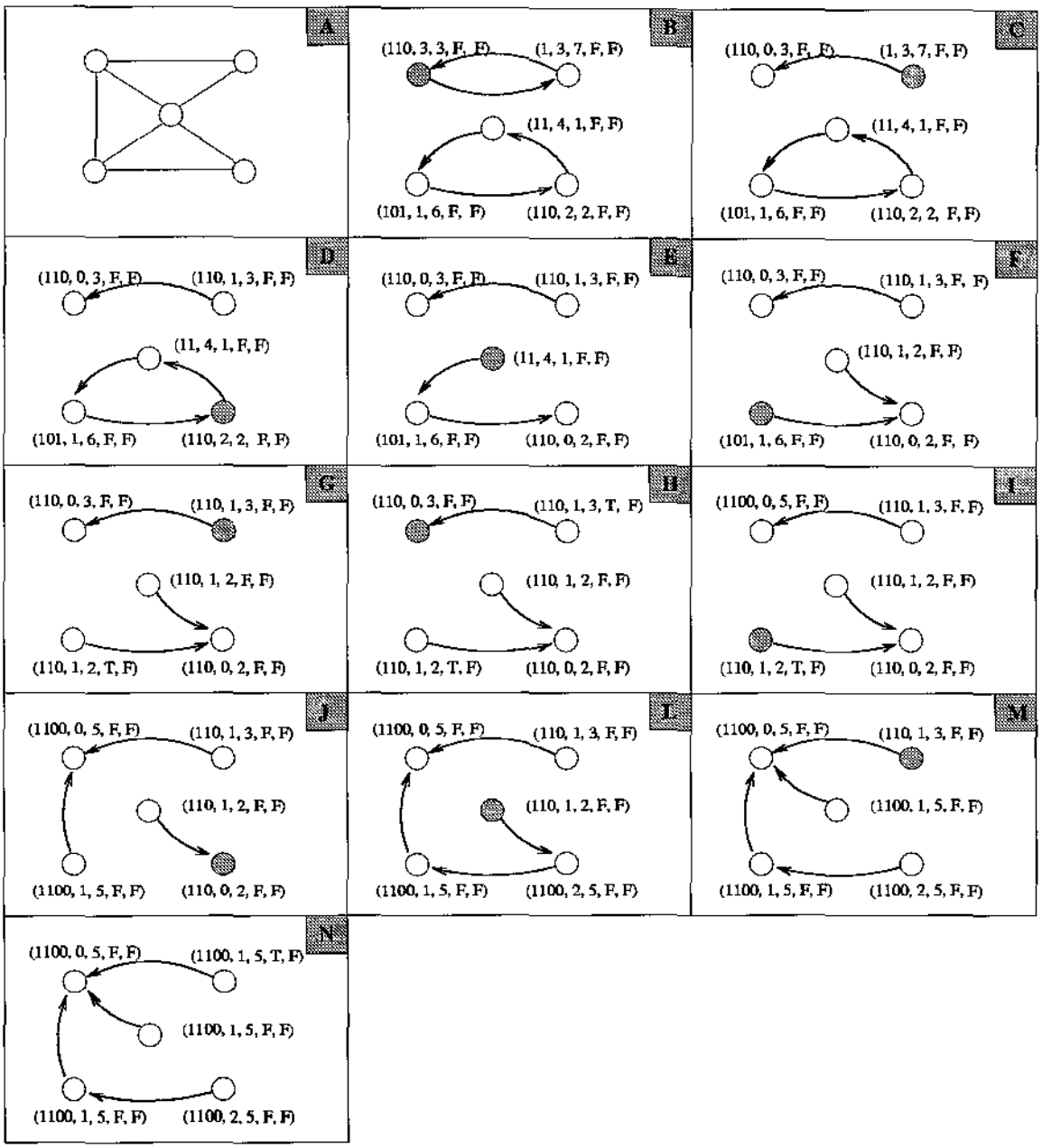


Figura 4.19: Exemplo de funcionamento do algoritmo auto-estabilizante para eleição em redes genéricas de Dolev et al.

Capítulo 5

Algoritmos Probabilísticos

As principais razões para o uso de algoritmos probabilísticos para eleição de líder são a necessidade de quebra de simetria, para o caso de redes anônimas, e o estabelecimento de probabilidades iguais de eleição a todos os processadores, para redes não anônimas. Devido à simetria (todos os processadores idênticos), não há algoritmo determinístico para eleição de líder em redes anônimas. O uso de probabilidade pode comprometer a correção ou a terminação do algoritmo, mas, no caso de redes anônimas, é uma forma de construir algoritmos para eleição. Para redes não anônimas, a maior parte dos algoritmos existentes elegem o processador de menor ou maior identificação como líder. No entanto os algoritmos descritos neste capítulo visam garantir a mesma probabilidade de se tornar líder a todos os participantes da eleição, independente de sua identificação.

Os algoritmos probabilísticos podem ser classificados em algoritmos Las Vegas e Monte Carlo de acordo com o grau de correção garantido [Tel94]. Um algoritmo Las Vegas é um algoritmo probabilístico que possui uma probabilidade p de terminar, mas sempre que termina garantidamente seu resultado é correto. Um algoritmo Monte Carlo é um algoritmo probabilístico que garantidamente termina, mas possui uma probabilidade $1 - p$ de seu resultado seja incorreto. Dentre os algoritmos descritos neste capítulo temos exemplos de algoritmos das duas categorias.

5.1 Algoritmos probabilísticos em redes anônimas

Nesta seção serão analisados dois tipos de algoritmos para redes anônimas: os que utilizam a informação de quantos nós existem na rede e os que não têm assegurado o tamanho da rede. O tamanho da rede é utilizado principalmente para ter conhecimento se uma mensagem já percorreu todos os processadores. A maior parte dos algoritmos descritos nesta seção utilizam anel unidirecional, no entanto é apresentado um algoritmo para redes genéricas, desenvolvido por Matias e Afek.

5.1.1 Com conhecimento do número de nós da rede

Nesta seção são apresentados dois algoritmos propostos em [IR90] para anéis anônimos com conhecimento do tamanho do anel, um síncrono e o outro assíncrono.

Modelo Síncrono

Nesta subseção é apresentado um algoritmo de Itai e Rodeh [IR90] para anel unidirecional uniforme e anônimo, utilizando números aleatórios para quebrar a simetria da rede. Também é descrita uma modificação a este algoritmo procurando diminuir o tempo necessário para a eleição.

Cada processador p_i da rede possui uma seqüência infinita de números reais aleatórios definida como $\{s[p_i,t]\}_{t=1}^{\infty}$ onde $0 \leq s[p_i,t] \leq 1$, fazendo com que a mudança de estado de p_i no *t_ésimo* intervalo de tempo dependa de $s[p_i,t]$.

O algoritmo é composto de fases de n intervalos de tempo cada. A cada fase temos $a \leq n$ processadores ativos. Durante uma fase alguns desses processadores tornam-se inativos, até que a seja reduzido a 1. Ao iniciar o algoritmo (intervalo de tempo $t = 0$) todos os processadores são ativos, ou seja $a = n$. No início de cada fase cada processador ativo torna-se um candidato a líder com probabilidade a^{-1}. Ao final da fase todos os processadores calcularam o número c de candidatos desta fase. Se $c = 1$ o único candidato torna-se líder. Se $c > 1$ então os candidatos da fase atual serão os processadores ativos da próxima fase. Se $c = 0$ esta fase foi inútil e os processadores ativos da próxima fase continuaram a ser os ativos da fase atual. Para que c seja calculado todos os processadores candidatos enviam uma mensagem no início da fase. Esta mensagem deve percorrer o anel e retornar a sua origem ao final da fase após n intervalos de tempo. Cada processador calcula c contando o número de mensagens que passaram por ele até o final da fase. Na figura 5.1 encontra-se uma descrição mais detalhada do algoritmo e na figura 5.2 temos um exemplo de funcionamento do algoritmo para $n = 6$, onde as letras A, C, I e L nos círculos que representam os processadores indicam se o processador é ativo, candidato, inativo ou líder respectivamente. No início da primeira fase ($t = 0$) todos os processadores estão ativos e alguns decidem tornarem-se candidatos. Os processadores candidatos da primeira fase tornam-se ativos na segunda fase ($t = n$) e dentre estes alguns serão os candidatos da segunda fase. Os que não foram candidatos na primeira fase tornam-se inativos. Como na terceira fase apenas um processador torna-se candidato este é eleito quando iria iniciar-se a quarta fase ($t = 3n$).

A probabilidade de se ter c candidatos dentre a processadores é denotada por $p(a,c)$. Como cada processador torna-se um candidato com uma probabilidade a^{-1} tem-se que

$$p(a,c) = \binom{a}{c} a^{-c}\left(1-\frac{1}{a}\right)^{a-c} = \left(1-\frac{1}{a}\right)^{a}\binom{a}{c}(a-1)^{-c},$$

sendo:

- $\binom{a}{c}$ o número de combinações de c elementos dentre um conjunto de tamanho a;

```
Algoritmo de Itai e Rodeh (A)
Variáveis:
    t = 0;
    ativo = TRUE;
    candidato = TRUE;
    a = n;
    c = n;
    fim = FALSE;
    líder = FALSE;

while not (fim) do begin
    receive M from p_{i-1};
    if (t mod n) ≠ 0 then begin
        if M ≠ nil then begin
            send (Mensagem) to p_{i+1};
            c = c + 1;
        end
    end
    else begin
        if c = 1 then begin
            if candidato then líder = TRUE;
            else líder = FALSE;
            fim = TRUE;
        end
        else begin
            if c > 1 then begin
                a = c;
                ativo = candidato;
            end
            if ativo and s[p_i,t] ≤ a^{-1} then begin
                candidato = TRUE;
                c = 1;
                send (message) to p_{i+1};
            end
            else begin
                candidato = FALSE;
                c = 0;
            end
        end
    end
end
```

Figura 5.1: Algoritmo síncrono para redes anônimas com n conhecido.

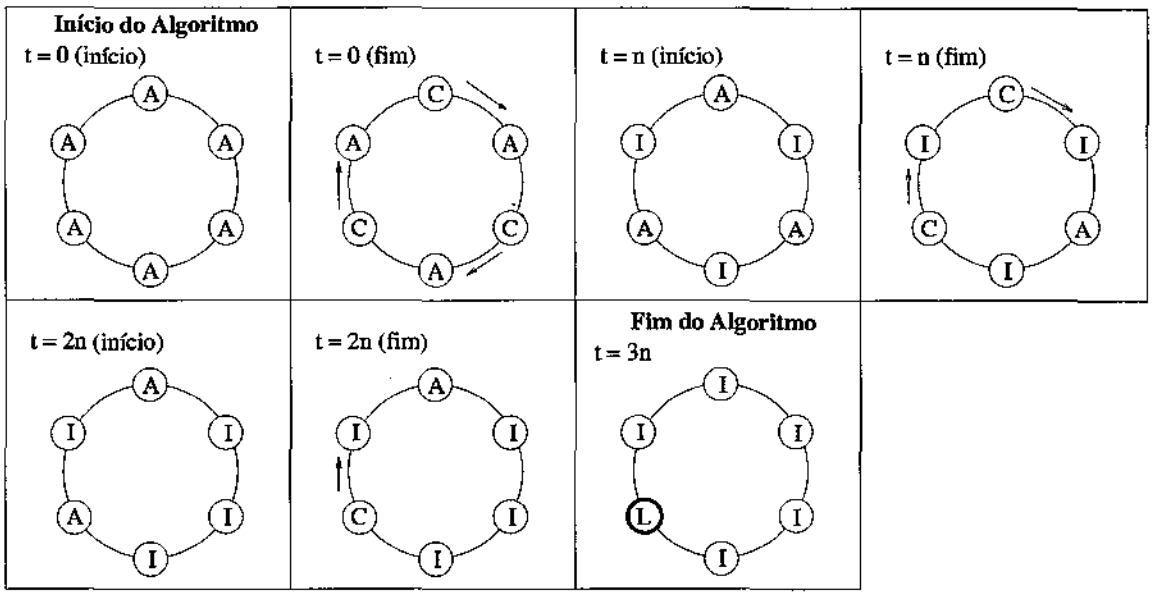


Figura 5.2: Exemplo de funcionamento do algoritmo síncrono de Itai e Rodeh com n conhecido.

- a^{-c} a probabilidade de se ter c candidatos;
- $\left(1-\frac{1}{a}\right)^{a-c}$ a probabilidade de ser ter $a-c$ não candidatos.

Conforme demonstrado em [IR90], o número de fases do algoritmo é $L \approx 2,44$ e o número médio de candidatos por fase é 1. Portanto, podemos concluir que a complexidade de tempo para eleger um líder em um anel síncrono com n processadores é igual a Ln. A complexidade de mensagens necessárias para a eleição é de Ln bits, pois cada fase possui n intervalos de tempo e na média um candidato, o qual envia uma mensagem que deve percorrer todos os n processadores do anel. Para este algoritmo são necessários $\log n$ bits de memória para armazenar o número de candidatos a líder em uma fase.

Uma modificação ao algoritmo anterior é proposta no artigo com a finalidade de diminuir o tempo de eleição. Essa alteração no algoritmo aumenta a complexidade de mensagens e a memória utilizada, adicionando um parâmetro q conhecido por todos os processadores. O algoritmo modificado também é dividido em fases e como anteriormente todos os processadores iniciam ativos. No início da fase, um processador envia uma mensagem com probabilidade q/n. Todo processador p possui um vetor $W(p)$ onde $W_i(p) = 1$ caso uma mensagem tenha passado por p no *i_ésimo* intervalo de tempo da fase. Os $W(p)$'s dos diferentes processadores são deslocamentos circulares uns dos outros, e a partir do seu $W(p)$ cada processador pode calcular os $W(p)$'s dos demais. Se todos os $W(p)$'s são distintos então o líder é escolhido como sendo o processador com $W(p)$ de menor valor lexicográfico. Caso contrário, o algoritmo deve ser executado novamente. Para esse melhoramento do algoritmo é demonstrado, no artigo original, que o número esperado de fases converge para $(1-e^{-q})^{-1}$ e a complexidade de mensagens em bits por processador converge para $q(1-e^{-q})^{-1}$.

Modelo Assíncrono

Ainda em [IR90] é proposto um algoritmo assíncrono, ou seja, semcontrole global de tempo que será descrito nesta subseção. O algoritmo constitui-se de duas fases: seleção e verificação. Inicialmente todos os processadores estão ativos. A fase de seleção é composta de c passos, sendo que um passo consiste em o processador escolher 0 ou 1, receber a mensagem de seu precedente e decidir se torna-se ou não inativo. Um processo inativo apenas transmite todas as mensagens por ele recebidas a seu vizinho no anel. Cada processador ativo executa o seguinte passo c vezes ou até tornar-se inativo: escolhe um valor 0 ou 1, cada um com probabilidade $\frac{1}{2}$ e envia a escolha ao próximo processador ativo. Um processador ativo tornar-se inativo caso sua escolha tenha sido 0 e o processador ativo que o precede tenha escolhido 1.

Ao terminar a fase de seleção pelo menos um processador permanece ativo, pois para tornar-se inativo o processador deve escolher 0, mas seu precedente deve escolher 1. No entanto, faz-se necessário verificar se apenas um processador permanece ativo (o processador líder), caso contrário deve ser executada novamente a fase de seleção. Na fase de verificação cada processador ativo envia um contador que deverá percorrer os n processadores do anel para certificar-se de que é o único ativo. O algoritmo é mostrado na figura 5.3. Uma iteração do algoritmo é exemplificada de maneira simples na figura 5.4, sendo que c_i representa o contador enviado pelo processador p_i que continua ativo ao final da fase e nc_i conta o número de mensagens contadores que p_i recebeu.

A seguir mostraremos uma razão intuitiva para a complexidade de mensagens ser de $O(n \log n)$ bits para $c = 5 \log n$. A probabilidade de um processador tornar-se inativo é de $\frac{1}{4}$ ($\frac{1}{2}$ para este processador escolher 0 e $\frac{1}{2}$ para seu precedente escolher 1). O número esperado de processadores inativos ao final do primeiro passo é de $n/4$, pois todos os n processadores ativos possuem probabilidade $\frac{1}{4}$ de tornarem-se inativos. Em geral, a cada passo um quarto dos processadores ativos tornam-se inativos, ou seja, a cada passo restam $\frac{3}{4}$ dos processadores ainda ativos. De acordo com as probabilidades, são necessários $\log_{4/3} n$ passos para que se tenha apenas um processador ativo. Escolhendo $c > 2 \log_{4/3} n \approx 4,81 \log n$ temos uma pequena probabilidade de que mais de um processador esteja ativo ao final dos c passos.

Como cada passo pode levar até n unidades de tempo para ser executado, pois uma mensagem pode ter que percorrer todo o anel, a complexidade de tempo da fase de seleção é $O(n \log n)$ e da fase de verificação é $O(n)$, totalizando um tempo igual a $O(n \log n)$.

O número de bits transmitidos a cada passo é n, já que cada processador envia apenas uma mensagem com o bit escolhido. Portanto, na fase de seleção a complexidade de mensagens é de $cn = O(n \log n)$ bits. Na fase de verificação cada um dos a processadores ativos enviam um contador que deve ter tamanho $\log n$ para percorrer o anel tendo então, um custo $an \log n$. Como é esperado que a seja 1 o custo é de $O(n \log n)$. Logo, a complexidade de mensagens do algoritmo é $O(n \log n)$ para as duas fases, com uma baixa probabilidade de repetição do algoritmo.

Algoritmo de Itai e Rodeh (B)

```
Variáveis:
    ativo = TRUE;
    iniciou = FALSE;
    meucont = 0;
    líder = FALSE;

// FASE DE SELEÇÃO
    Inicia();
    while not fim do begin
        receive M from p_{i-1};
        if M = VALOR(valor) then
            if not ativo then
                send VALOR(valor) to p_{i+1};
            else
                if valor = 1 and meuvalor = 0 then
                    ativo = FALSE;
                else
                    if c > 1 then begin
                        Escolhe(meuvalor); // 0 ou 1
                        send VALOR(meuvalor) to p_{i+1};
                        c = c - 1;
                    end
                    else begin
                        // INÍCIO DA FASE DE VERIFICAÇÃO
                        cont = 1;
                        send CONTADOR(cont) to p_{i+1};
                        meucont = 0;
                    end
        else if M = CONTADOR(cont) then
            if ativo then begin
                meucont = meucont +1;
                if cont = n then
                    if meucont = 1 then begin
                        líder = TRUE;
                        send ELEITO to p_{i+1};
                    end else
                        // REINICIA FASE DE SELEÇÃO
                        Inicia();
                else begin
                    cont = cont + 1;
                    send (cont) to p_{i+1};
                end
            else begin
                cont = cont +1;
                send (cont) to p_{i+1};
            end
        else if M = ELEITO then begin
            if not líder then
                send ELEITO to p_{i+1};
            fim = TRUE;
        end
    end

Procedure Inicia()
begin
    iniciou = TRUE;
    Escolhe (meuvalor); // 0 ou 1
    send (meuvalor) to p_{i+1};
    c = 5 log n;
end
```

Figura 5.3: Algoritmo assíncrono para redes anônimas com n conhecido.

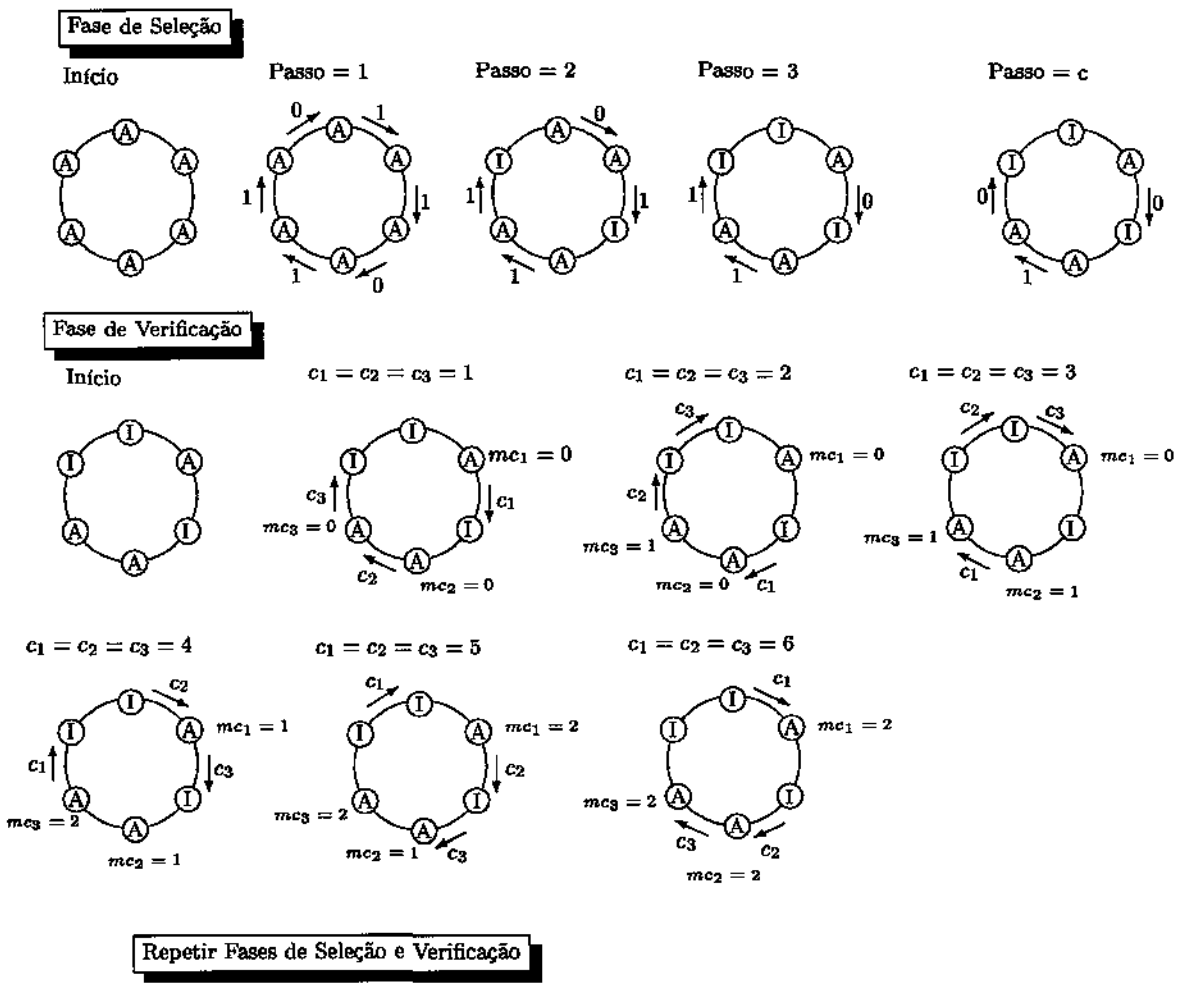


Figura 5.4: Exemplo de uma iteração do algoritmo assíncrono de Itai e Rodeh com conhecimento de n.

5.1.2 Sem conhecimento sobre o tamanho da rede

Nesta subseção apresentamos os algoritmos para eleição de líder em redes anônimas onde não se conhece o número n de nós da rede. Como demonstrado em [IR90], não há algoritmo com detecção de terminação para descobrir o tamanho da rede que tenha probabilidade de erro menor que 1. Portanto, nesta seção serão apresentados algoritmos que usam o conhecimento dos limites de n e um algoritmo onde a terminação não pode ser detectada.

Modelo Síncrono

O algoritmo proposto por Itai e Rodeh [IR90] apresentado nesta subseção utiliza anel uniforme unidirecional com conhecimento dos limites de n, $N \leq n < 2N$. Este algoritmo possui duas fases: seleção e verificação.

A fase de seleção pode ser composta de vários passos, cada uma com duração de $2N-1$ intervalos de tempo, até que um ou mais candidatos a líder sejam escolhidos. No primeiro intervalo de tempo do passo cada processador escolhe um $valor \in \{0,1\}$, sendo que a probabilidade de se escolher 1 é q/n. Um processador que escolha o valor 1 é chamado de candidato. Cada candidato deve então enviar uma mensagem no primeiro intervalo de tempo de cada passo. Durante os $2N-1$ intervalos de tempo restantes para o término do passo toda mensagem recebida por um processador que não seja candidato deve ser retransmitida. Se nenhum processador escolher 1, passarão os $2N-1$ intervalos de tempo e nenhuma mensagem será transmitida, portanto todos terão conhecimento de que não houve seleção de nenhum candidato a líder e um novo passo da fase de seleção será executado. Caso contrário será executada a fase de verificação.

Duas sub-fases compõem a fase de verificação. Durante a primeira sub-fase os processadores candidatos enviam uma mensagem no primeiro intervalo de tempo da sub-fase, mas não retransmitem as mensagens recebidas; os demaisprocessadores apenas retransmitem todas as mensagem que recebem. Esta sub-fase tem duração de $N-1$ intervalos de tempo. Se um único candidato foi escolhido, uma única mensagem será enviada e esta não percorrerá todo o anel durante esta sub-fase, pois $n > N-1$. Logo o candidato não receberá mensagem alguma. No entanto, se há mais de um candidato, sendo $n \leq 2N-1$, existem dois candidatos com distância $\leq N-1$ entre eles, e pelo menos um tomará conhecimento de existência de mais de um candidato. Na segunda sub-fase de verificação, apenas os candidatos que sabem que existe mais de um candidato enviam mensagem e não retransmitem as mensagens recebidas, os demais processadores apenas retransmitem as mensagem que chegam até eles. Esta sub-fase tem duração $2N-1$ para assegurar que se houver mais de um candidato todos os processadores do anel terão conhecimento deste fato e reiniciarão uma nova eleição. Caso haja apenas um candidato no anel nenhuma mensagem será enviada na segunda sub-fase de verificação, portanto todos saberão que há um único candidato e este será o líder. A descrição mais detalhada do algoritmo é mostrada na figura 5.5.

A cada invocação do algoritmo a complexidade de mensagens em bits da fase de seleção é de n bits, pois, apesar de poderem existir vários passos na fase de seleção, somente quando um ou mais candidatos são escolhidos, ou seja no último passo, é que há comunicação. Na primeira sub-fase de verificação são gastos $N-1$ bits em mensagens caso haja apenas um candidato,

Algoritmo de Itai e Rodeh (C)

```
Variáveis:
  t = 0;
  fase = 1;
  sub_fase = 0;
  nmsg = 0;
  candidato = FALSE;
  fim = FALSE;

  while not fim do begin
    if fase = 1 then begin
      FASE DE SELEÇÃO
      if t = 0 then begin
        Escolhe(id);
        if id = 1 then begin
          candidato = TRUE;
          send (msg) to p_{i+1};
        end
        else
          candidato = FALSE;
      end
      else begin
        receive (msg) from p_{i-1};
        if t < 2N - 2 then begin
          if msg ≠ nil and not candidato then
          begin
            nmsg = nmsg + 1;
            send (msg) to p_{i+1};
          end
        end
        else begin
          if msg ≠ nil then
            nmsg = nmsg + 1;
          if nmsg > 0 or candidato then begin
            fase = 2;
            sub_fase = 1;
            t = 0;
          end
          else
            t = 0;
        end
      end
    end
    else if fase = 2 then begin
    FASE DE VERIFICAÇÃO
      if sub_fase = 1 then
      begin
        if t = 0 then begin
          if candidato then
            send (msg) to p_{i+1};
          nmsg = 0;
        end
        else begin
          receive (msg) from p_{i-1};
          if msg ≠ nil then
              if candidato then
                nmsg = nmsg + 1;
              else send (msg) to p_{i+1};
            if t = N - 1 begin
              sub_fase = 2;
              t = 0;
            end
        end
      end
      else if sub_fase = 2 then begin
        if t = 0 then begin
          if candidato and nmsg > 0 then
            send (msg) to p_{i+1};
        end
        else begin
          receive (msg) from p_{i-1};
            if msg ≠ nil then begin
              send (msg) to p_{i+1};
              nmsg = nmsg +1;
            end
            if t = 2N - 1 then begin
              if nmsg = 0 then begin
                if candidato then líder = TRUE;
                else líder = FALSE;
                fim = TRUE;
              end else begin
                fase = 1;
                t = 0;
                nmsg = 0;
              end
            end
          end
        end
      t = t + 1;
    end
```

Figura 5.5: Algoritmo síncrono para redes anônimas com $N \leq n < 2N$.

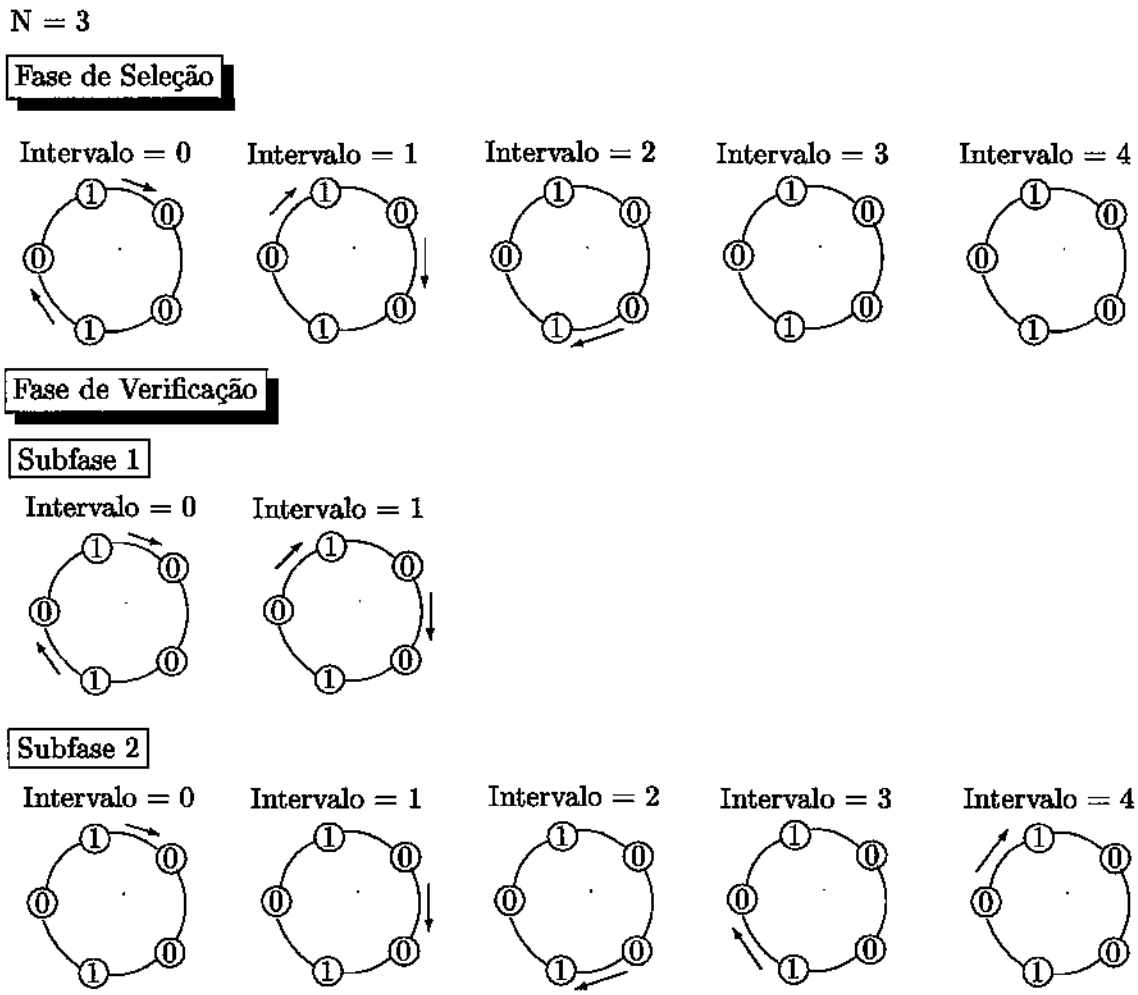


Figura 5.6: Exemplo de uma iteração do algoritmo síncrono de Itai e Rodeh com $N \leq n < 2N$.

ou no máximo n bits caso contrário. Na segunda sub-fase da fase de verificação são gastos n bis se houver mais de um candidato e nenhum bit caso contrário. Portanto a cada invocação do algoritmo são gastos $n + N - 1$ bits se um único candidato for selecionado, caso contrário são necessários $\leq 3n$ bits. A complexidade de tempo de cada invocação do algoritmo é de $k_{sel}(2N-1)+3N-2$, sendo k_{sel} o número de repetições da fase de seleção até que pelo menos um candidato seja escolhido. O número k_{sel} e a probabilidade de que o algoritmo seja invocado mais de uma vez (mais de um candidato na fase de seleção) dependem de q, N e n. De acordo com demonstrações encontradas em [IR90] o número esperado de iterações do algoritmo todo é $k_{alg} \approx 4,27$ e $k_{sel} \approx 2,54$. Portanto a complexidade de tempo total do algoritmo é:

$$T = k_{alg}(3N - 2 + k_{sel}(2N-1)) < 34,52N - 19,39.$$

Modelo Assíncrono

Nesta subseção são apresentados três algoritmos: o de Itai e Rodeh [IR90] (uma modificação ao algoritmo da seção 5.1.1), o algoritmo proposto por Matias e Afek [MA89, AM94] onde nada se sabe sobre n e o de Itai [Ita90] que preocupa-se com o espaço de armazenamento necessário.

Em [IR90], o algoritmo proposto para redes anônimas com $N \leq n < 2N$ consiste no algoritmo assíncrono da seção 5.1.1 com uma modificação na fase de verificação, onde a mensagem deverá percorrer uma distância de $2N-1$, ao invés de n, para assegurar que esta passou por todos os processadores do anel pelo menos uma vez, pois n não é conhecido. Caso algum processador receba mais de duas mensagens na fase de verificação este terá conhecimento de que mais de um líder foi escolhido, devendo passar esta informação aos outros para que uma nova eleição seja iniciada. A complexidade de mensagens em bits do algoritmo é de $O(n \log N + (2N-1)\log N)$, ou seja, $O(n \log n)$ bits.

Em [Lav95] encontramos a análise do um algoritmo proposto por Matias e Afek em [MA89, AM94]. Este é um algoritmo simples para eleição em rede anônima sem nenhum conhecimento sobre topologia e tamanho, mantendo a eficiência da complexidade de mensagens. O algoritmo possui uma probabilidade de erro $\epsilon < 1$, sendo ainda uma forma eficiente de construir uma árvore geradora da rede. O algoritmo é dividido em dois passos. Primeiramente, cada processador escolhe uma identificação id aleatoriamente. Em seguida, essa identificação deve ser difundida pela rede. Ao receber uma identificação id, cada processador deve verificar se nenhuma identificação maior ou igual a esta já foi recebida. Caso esta seja a maior, o processador deve repassá-la a todos os seus vizinhos e marcar a origem de tal mensagem como sendo seu pai (pai = *origem da mensagem*), construindo assim a árvore geradora. Caso contrário, a mensagem é ignorada. Ao final todos conhecerão a maior identificação id na rede e teremos uma árvore geradora cuja raiz é o líder.

O ponto principal do algoritmo é a escolha de id de tal forma que a probabilidade de se ter uma única identificação máxima e poucos $id's$ diferentes seja alta. A identificação id_i de um processador P_i deve ser composta por dois números t_i e s_i. Para escolher t_i P_i lança uma moeda não viciada até obter *Cara* pela primeira vez, e t_i é o número de lances realizados. O número s_i

é escolhido aleatoriamente dentro do intervalo $[1..d]$, onde $d = O(r \log r)$ e $r = \frac{1}{\epsilon}$. Na figura 5.7 o algoritmo é mostrado de forma detalhada.

```
Algoritmo de Matias e Afek
Variáveis:
    fim = FALSE;

    Escolhe(meuid);
    max = meuid;
    pai = 0;
    send (meuid) to todos os vizinhos;
    while not fim do begin
        receive M from c;
        if M = ID(id) then begin
            if id > max then begin
                max = id;
                pai = c;
                send (id) to todos os vizinhos;
            end
        end
    end

Procedure Escolhe(id)
begin
    t = 1;
    lance a moeda;
    while moeda = Coroa do
    begin
        lance a moeda;
        t = t + 1;
    end
    s = random(1,d);
    id = (t,s);
    return id;
end
```

Figura 5.7: Algoritmo de Matias e Afek para redes anônimas sem conhecimento de n.

Para este algoritmo não há detecção de terminação, pois nada se conhece sobre o tamanho da rede. No entanto, é garantido que em algum momento todos os processadores da rede terão o mesmo máximo, o sistema estará estável e o líder será o processador com $pai = 0$.

A figura 5.8 podemos ver um exemplo de funcionamento do algoritmo de Matias e Afek. No exemplo a identificação máxima que cada processador conhece é representada por apenas um número e apresentada dentro do círculo que representa o processador. Sempre que um processador inicia a execução, que acontece espontaneamente ou ao receber uma mensagem, ele escolhe um valor para sua identificação e a envia a todos os seus vizinhos. Inicialmente (estágio A) somente dois processadores iniciam a eleição. Ao final (estágio E) a maior identificação a circular pela rede é a igual a 7 e uma árvore geradora do grafo que representa a topologia da rede é formada.

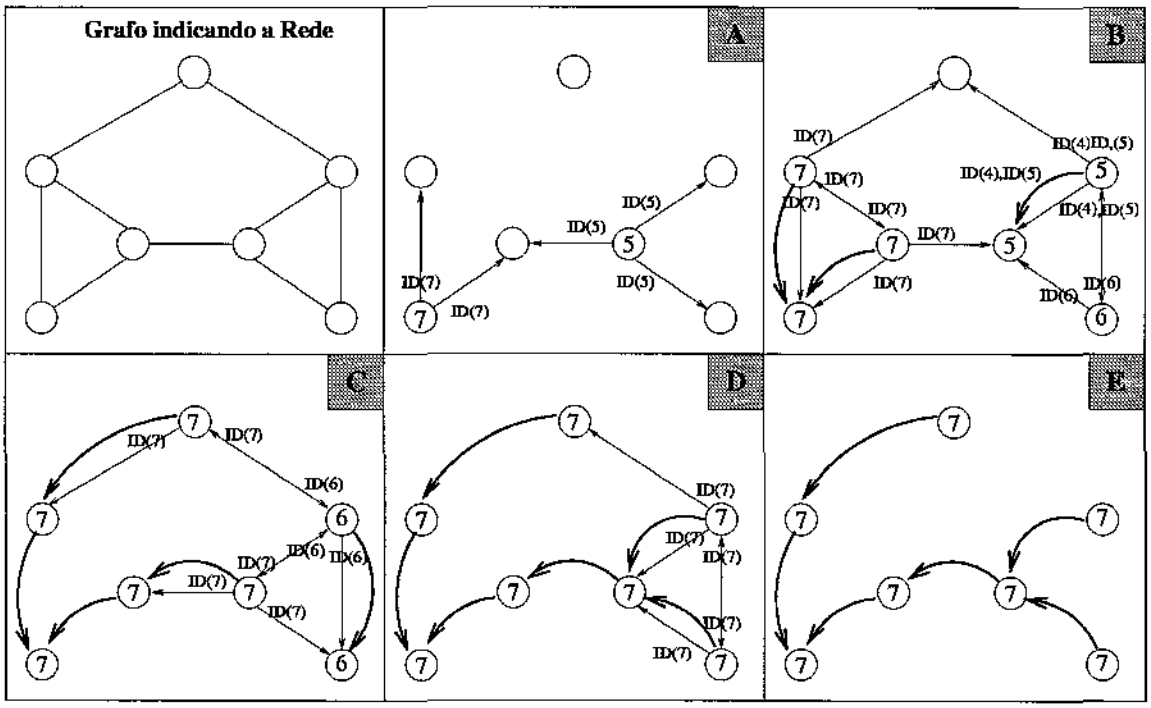


Figura 5.8: Exemplo de funcionamento do algoritmo de Matias e Afek para redes anônimas genéricas.

Em [MA89] é demonstrado que o algoritmo garante os itens abaixo:

1. Apesar de não haver detecção de terminação, em algum momento todos os processadores terão o mesmo máximo, que corresponde à maior identificação dentre todas as escolhidas pelos processadores. A partir deste momento nenhuma mensagem será gerada e o algoritmo pára.

2. Usando o método definido para a escolha das identificações e com a probabilidade de erro ϵ, uma única identificação com valor máximo é gerada. Portanto, teremos um único líder na rede e a construção da árvore geradora.

3. Com base na probabilidade de erro ϵ, a complexidade de mensagens é de $O(mr \log n \log r)$ (m é o número total de canais de comunicação), onde cada mensagem tem tamanho $O(\log \log n + \log r)$ bits. A complexidade de tempo é $O(D)$, onde D é o diâmetro da rede.

Outro algoritmo assíncrono para redes anônimas que usa probabilidade é o apresentado por Itai em [Ita90], que considera os processadores organizados em um anel unidirecional com tamanho desconhecido. Este algoritmo usa algumas idéias do algoritmo de Matias e Afek [MA89, AM94], como escolher um número aleatório e vencer quem tiver o maior número. Um processador pode mudar seu estado de acordo com uma mensagem recebida, ou seja, pode ser que um

processador que dizia-se líder, mude para não líder dependendo da mensagem. As mensagens que são enviadas serão recebidas em um tempo finito e a ordem FIFO é mantida nos canais de comunicação.

A grande preocupação do autor nesse artigo foi em relação ao espaço de armazenamento, procurando utilizar uma memória limitada e de tamanho $O(1)$, ou seja, complexidade de espaço igual a $O(1)$. No entanto o algoritmo possui uma probabilidade de erro igual a $O(1/n)$ no resultado da eleição, pois é possível que mais de um processador seja eleito.

A mensagem que inicia o algoritmo é designada *iniciar*. Os processadores que recebem uma mensagem *iniciar* são chamados *reis*. Cada rei escolhe um identificador *id* aleatório, que é enviado através do anel em uma mensagem $M(id)$. O tamanho de *id* também é aleatório, havendo uma probabilidade 2^{-i} de que o tamanho seja i.

Ao receber uma mensagem $M(mid)$, o rei compara *mid* com seu próprio *id*. Se $mid > id$, a mensagem continua através do anel. Se $mid < id$, a mensagem não é mais passada adiante. E se $mid = id$ então o rei assume que seu próprio *id* retornou, ele possui a maior identificação e deve ser nomeado líder. No entanto, esta suposição pode ser errada, pois um outro processador pode ter escolhido o mesmo *id* e é possível que haja mensagens maiores que ainda não chegaram. Portanto, o líder continua participando da eleição e conseqüentemente pode vir a deixar o estado de líder.

O problema do espaço de armazenamento ser limitado é resolvido da seguinte forma: se o *id* escolhido por um rei p_r é o número binário $[a_1, ..., a_l]$, p_r armazena seu identificador em um conjunto de processadores $R_r = \{p_r, ..., p_{r+l}\}$, que é chamado de seu reino, isto é, a_1 é armazenado no processador p_r, a_2 em p_{r+1},..., a_l em p_{r+l-1}. O último processador do reino, p_{r+l} , possui o símbolo "#"indicando que este processador é o último do reino. Para simplificar, mostraremos aqui apenas a versão do algoritmo que considera reinos disjuntos. Note que a comparação dos identificadores é feita de forma distribuída, cada processador do reino compara a parte que ele possui do identificador com a mensagem que receber.

Quando o reino $R_r = \{p_r, ..., p_{r+l}\}$ obtém a mensagem $id' = [b_1, ..., b_q]$ ele a compara com seu próprio *id*. Se os identificadores são iguais e nenhum *id* maior é conhecido, então o último processador do reino, p_{r+l}, torna-se líder se $id < id'$ (isto é, $l < q$ ou $l = q$ e $id < id'$ enquanto números binários, ou seja, $\sum a_r 2^{l-r} < \sum b_r 2^{l-r}$) então id' continua a passar pelo anel. Assim o *id* máximo somente poderá parar quando retornar ao seu originador.

As variáveis usadas no algoritmo pelo processador p_i são basicamente duas id_i que contém o bit i da identificação gerada pelo reino ou o valor # indicando que é o último processador do reino e mem_i (0,1 ou #) utilizada para a transmissão das mensagens. Na descrição detalhada do algoritmo apresentada na figura 5.9, temos ainda algumas variáveis de controle: *transmite* (usada pelo último processador do reino para saber se é a primeira mensagem $TRANSMITE$ recebida), *investiga* (usada para que um processador saiba se já recebeu alguma mensagem $INVESTIGA$ ou não), *ganhou* (indica se o reino ganhou a comparação) e *líder* (indica que o processador encontrou uma identificação igual a sua e acredita ser o líder). As mensagens enviadas são: $ESCOLHE$ (inicialmente para escolher a identificação do reino), $TRANSMITE$ (para que a identificação percorra o reino para que seja transmitida pelo anel), $INVESTIGA$

(transmite a identificação de um reino a outro para que seja realizada a comparação das identificações) e *COMPARA* (percorre o reino comparado os bits da identificação).

Como vimos a memória interna necessária a cada processador p_i é de tamanho constante, assim como as mensagens. No entanto o número de mensagens é alto, pois cada mensagem transmite $O(1)$ bit e a identificações dos reinos de tamanho $O(\log n)$ devem percorrer todo o anel.

O tamanho máximo esperado para *id* é $\log n$ e temos uma probabilidade $p > 1 - 1/n$ de que o maior *id* seja único (lembre-se da probabilidade de erro do resultado igual a $1/n$). Suponha que k processadores recebam a mensagem *iniciar*, é esperado que cada um dos k processadores escolham *id's* de tamanho $O(\log k)$. Portanto a complexidade de mensagens esperada é $O(kn \log k)$, ou seja, $O(n^2 \log n)$ bits.

5.2 Algoritmos probabilísticos em redes não anônimas

A seguir, nesta seção, são mostrados algoritmos que garantem probabilidades iguais de vitória a todos os processadores que participam da eleição, independente de seus identificadores. Serão apresentados algoritmos propostos em [Cha90] e [VAD94].

No artigo [Cha90], Chang trata da importância de se escolher um líder dando uma mesma probabilidade de vitória a todos os competidores. O esquema proposto é baseado em uma *licitação* onde cada participante envia a todos os outros um valor escolhido aleatoriamente. O processador de eleição consiste em decidir quem será o líder baseado nos valores escolhidos. É demonstrado que o algoritmo de eleição, juntamente com o intervalo de números possíveis, garante que todo participante tem probabilidade $1/n$ de ser eleito. Um outro artigo [VAD94] apresenta a mesma solução básica de [Cha90] para garantir probabilidade igual para todos os participantes de vencer.

O algoritmo não garante *justiça* (isso não quer dizer que não esteja correto) no caso de existir processador malicioso. O problema é que um processador pode *trapacear* e sempre garantir sua eleição, bastando para isso esperar receber as escolhas de todos os outros processadores antes de fazer sua escolha. Para um algoritmo de eleição de líder estar correto deve-se garantir que um e apenas um processador seja eleito. No caso deste algoritmo com a aplicação da função (mod n) isto é garantido.

Como a preocupação do algoritmo é garantir que todos os processadores tenham a mesma chance de vencer, o custo nem é discutido no artigo, mas como todos os processadores devem enviar mensagens a todos os outros a complexidade de mensagem é $O(n^2)$, um custo pior em relação a várias das soluções estudadas.

Considere uma rede não anônima com um conjunto $S = \{P_0, P_1, ..., P_{n-1}\}$ de processadores, onde a comunicação é realizada através de troca de mensagens. Em [Cha90] e [VAD94] apresentam o mesmo algoritmo básico, onde para escolher um líder entre os n processadores, cada um deve selecionar um valor para competir e enviá-lo aos demais processadores. Ao receber os valores escolhidos pelos outros processadores, cada processador, para descobrir quem venceu,

Algoritmo de Itai para o Processador p_i

```
transmite = FALSE;
investiga = FALSE;
ganhou = FALSE;
líder = FALSE;
if receive iniciar from p_{i-1} then begin
  rei = TRUE;
  id_i = Escolhe();
  send ESCOLHE(id_i) to p_{i+1};
  send TRANSMITE("fim") to p_{i+1};
end else rei = FALSE;
while TRUE do begin
  receive M from p_{i-1};
  case M of
    ESCOLHE(a):
      mem_i = a;
      if decide ser o último then id_i = "#";
      else begin
        id_i = Escolhe();
        send ESCOLHE(id_i) to p_{i+1};
      end
    TRANSMITE(a):
      if id_i = "#" then begin
        if not transmite then begin
          send INVESTIGA("#") to p_{i+1};
          transmite = TRUE;
        end
        send INVESTIGA(mem_i) to p_{i+1};
        if a ≠ "fim" then mem_i = a;
        else send INVESTIGA("fim") to p_{i+1};
      end
      else begin
        send TRANSMITE(mem_i) to p_{i+1};
        if a ≠ "fim" then mem_i = a;
        else send TRANSMITE("fim") to p_{i+1};
      end
    INVESTIGA(a):
      if rei and a = "fim" then begin
        if mem_i = id_i then
          send COMPARA("=") to p_{i+1};
        else if mem_i ¿ id_i then
          send COMPARA("-") to p_{i+1};
        else
          send COMPARA("+") to p_{i+1};
        send INVESTIGA(mem_i) to p_{i+1};
      end else begin
        if not investiga then begin
          mem_i = a; end
          investiga = TRUE;
        end else if not ganhou begin
          send INVESTIGA(mem_i) to p_{i+1};
          mem_i = a;
        end
      end
    COMPARA(a):
      if id_i = "#" then begin
        if a ≠ "fim" then
          if a = "=" then begin
            if id_i = mem_i then begin
              líder = TRUE;
              ganhou = TRUE;
            end else begin
              líder = FALSE;
              ganhou = FALSE;
            end
          end else if a = "+" then
            if id_i = mem_i then
              ganhou = TRUE;
            else begin
              líder = FALSE;
              ganhou = FALSE;
            end
        else ganhou = TRUE;
      else
        if mem_i = "#" then
          send COMPARA("fim") to p_{i+1};
        else
          if a = "=" then begin
            if mem_i = id_i then
              send COMPARA("=") to p_{i+1};
            else if mem_i > id_i then
              send COMPARA("-") to p_{i+1};
            else
              send COMPARA("+") to p_{i+1};
          else
            send COMPARA(a) to p_{i+1};
end
```

Figura 5.9: Algoritmo de Itai para anel anônimo.

aplica uma função sobre os n valores (o seu e os $n-1$ recebidos) e o número k resultante da função indica o processador P_k escolhido.

Seja B o conjunto $\{0,...,n-1\}$ do qual os processadores escolhem seus valores para a competição e b_i o valor selecionado pelo processador P_i. A função é definida por $f(b_0,...,b_{n-1}) = k = \sum_{i=0}^{n-1} b_i \bmod n$.

A terminação do algoritmo ocorre após todos receberem as mensagens dos demais processadores e assim encontrarem a identificação do vencedor. Para esta solução, independente da identificação do processador e dos valores escolhidos para a competição, cada processador possui uma probabilidade igual a $1/n$ de vencer. O cardinalidade de B usada nesta solução é mínima, pois para garantir a mesma probabilidade a todos devemos ter $|B| \geq n$. Portanto o tamanho da mensagem é $\log |B|$ bits, pois cada uma contém apenas o valor escolhido por um processador. Este algoritmo é descrito na figura 5.10. Uma pequena ilustração do funcionamento do algoritmo básico é apresentado na figura 5.11.

Algoritmo de Chang ou Vernadat et al.

```
for (c = 0; c < n; c++) do
      recebi[c] = FALSE;
Escolhe(b[i]);
recebi[i] = TRUE;
while(∃ c | recebi[c] = FALSE) do begin
      receive valor from c;
      b[c] = valor;
      recebi[c] = TRUE;
end
if ( i = (∑_{c=0}^{n-1} b[c]) mod n) then
      líder = TRUE;
else
      líder = FALSE;
```

Figura 5.10: Algoritmo básico de Chang ou Vernadat et al.

Algumas modificações ao algoritmo básico são apresentadas tanto em [Cha90] como em [VAD94]. Em [Cha90] é apresentada uma modificação para que alguns processadores tenham probabilidade maior de vencer que outros. Para isso são acrescentados pseudo-processadores, P_n a P_{m-1}, ao conjunto S. Caso P_i deva ter k vezes a probabilidade de vencer, associamos $k-1$ pseudo-processadores ao processador P_i. Portanto, $m-n$ corresponde ao número total de pseudo-processadores necessários para que as probabilidades de vitória dos processadores sejam ampliadas.

Cada processador real deve escolher um valor do conjunto $B = \{0,...,m-1\}$, e cada pseudo-processador vai usar o mesmo valor escolhido por seu processador real correspondente. A nova função é definida por $f(b_0,...,b_{m-1}) = k = \sum_{i=0}^{m-1} b_i \bmod m$. Se P_k for um pseudo-processador o vencedor é o processador real associado a ele. Na figura 5.12 temos um exemplo para o caso de alguns processadores possuírem mais chances, onde os pseudo-processadores são apresentados

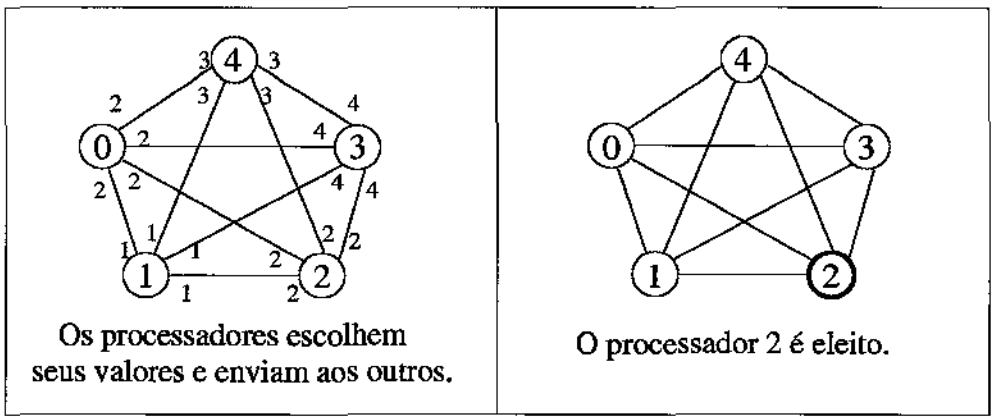


Figura 5.11: Exemplo de funcionamento do algoritmo básico de Chang ou Vernadat et al.

na mesma cor de seu processador real associado, sendo $n = 5$ e $m = 7$.

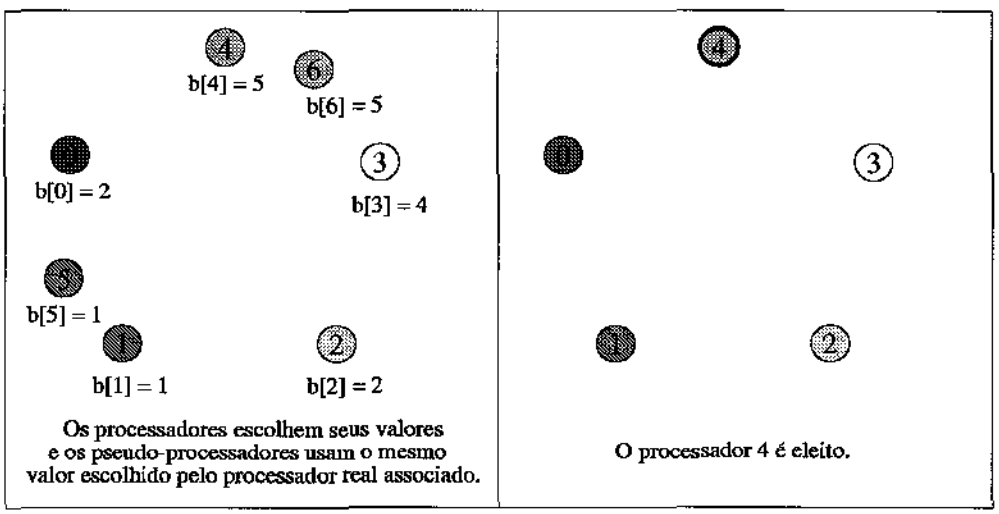


Figura 5.12: Exemplo de funcionamento do algoritmo de Chang com alguns processadores possuindo maiores probabilidades de vencer. Os pseudo-processadores estão representados na mesma cor e próximos dos processadores reais a que estão associados.

Outras situações nas quais mais de um processador deve ser selecionado durante uma competição podem existir. Para estes casos mais uma modificação ao algoritmo básico é proposta por Chang a fim de escolher m vencedores. Cada processador P_i deve escolher m valores, $b_i^0, ..., b_i^{m-1}$, onde para $0 \leq k \leq m-1$, b_i^k é escolhido do conjunto $\{0, ..., n-k-1\}$. Em seguida o processador envia os valores selecionados aos demais processadores.

Após receber os valores escolhidos por todos os $n-1$ processadores, cada processador deve aplicar m funções $f^0, ..., f^m - 1$ para que os m processadores sejam selecionados. Ao escolher o $k+1$-ésimo vencedor, para k de 0 a $m-1$, deve ser aplicada a função $f^k(b_0^k, ..., b_{n-k-1}^k) = j = \sum_{i=0}^{n-k-1} b_i^k \bmod (n-k)$, sendo P_j o vencedor. Então, deve-se substituir P_j por P_{n-k-1} para o

cálculo do restante dos vencedores.

No caso em que o número n total de potenciais competidores é conhecido, porém nem todos realmente participarão da eleição, o procedimento apresentado em [Cha90] é o seguinte. Cada processador P_i participante da competição escolhe $n-1$ valores, $b_i^0, ..., b_i^{n-2}$, onde b_i^k, para $0 \leq k \leq n-2$, é selecionado do conjunto $\{0, ..., n-k-1\}$. Em seguida P_i envia os $n-1$ valores aos outros processadores. Passado um intervalo de tempo determinado para que os valores escolhidos por todos os processadores participantes sejam recebidos, P_i deve contar quantos participantes da competição existem. Seja m o número de real competidores, cada processador deve então renumerar os processadores de Q_0 a Q_{m-1}, sendo Q_j o j-ésimo processador com menor identificação dentre os m participantes. Então é aplicada a função $f(b_0^{n-m}, ..., b_{m-1}^{n-m}) = k = \sum_{i=0}^{m-1} b_i^{n-m} \bmod (m)$. O processador correspondente a Q_k é o vencedor.

Em [VAD94] a solução para o caso de m competidores dentre n processadores é diferente. A proposta consiste em mudar o intervalo de escolha dos valores para $[1, ..., n!]$. Cada processador que deseja participar da competição escolhe um valor do intervalo e envia a todos os outros. Após esperar as mensagens de todos os competidores, cada processador deve reordenar os competidores como no algoritmo de Chang do parágrafo anterior. Então é aplicada a função (mod m) sobre a soma dos m valores conhecidos pelos competidores e o processador que corresponde ao resultado é o vencedor. Com o intervalo de $[1, ..., n!]$ é garantido que todos os processadores possuirão a mesma probabilidade de se tornar líder.

Capítulo 6

Conclusão

Durante o desenvolvimento deste trabalho de mestrado foi coletado um extenso material bibliográfico na área. No entanto, devido à limitação de tempo, foi necessário selecionar alguns trabalhos a fim de serem estudados mais profundamente.

Dentre os algoritmos clássicos, apresentamos um dos primeiros algoritmos de eleição que se tem conhecimento, proposto por LeLann em [LeL77]. A partir deste trabalho surgiram outros, como [CR79] e [Pet82], propondo melhoramentos, e que também usavam a organização dos processadores em um anel uniforme. O algoritmo de LeLann visa a composição de uma lista de processadores ativos em cada nó, acarretando um alto custo de $O(n^2)$ mensagens tanto no melhor como no pior caso. Chang e Roberts conseguiram melhorar este custo para $O(n \log n)$ no caso médio e $O(n)$ no melhor caso, eliminando as mensagens dos processadores com identificadores menores. O algoritmo apresentado por Peterson, que utiliza a idéia de simulação de bidirecionamento, consegue melhorar o custo de mensagens para $O(n \log n)$ no pior caso.

O algoritmo de Garcia-Molina, também uma solução clássica para o problema trabalhando com topologia completa, tem como grande preocupação a falha do processador coordenador que deve ser substituído pelo processador com maior identificação ainda em atividade, tão logo fosse detectada a falha. O custo do algoritmo é maior quando o processador que detecta a falha possui um valor baixo para sua identificação, o que acarreta que todos os processadores com identificação maior que a dele irão tentar eleger-se. Assim, o custo de mensagens, no pior caso, é $O(n^2)$.

Um conceito interessante que tem recebido atenção nos últimos anos é o uso de *senso de direção* [FM96, Tel95, FMS96, FMS]. O intuito de usar canais de comunicação rotulados de acordo com uma propriedade global é quebrar a simetria da rede e diminuir o número de mensagens devido ao conhecimento que se adquire com os rótulos. Em [FMS] é apresentada uma definição formal para senso de direção e identificadas quatro classes de rótulos para os processadores. Nesta dissertação apresentamos alguns algoritmos que utilizam esta abordagem. Os algoritmos de Loui *et al* [LMW86], um dos primeiros algoritmos de eleição que usa este conceito, e de Singh [Sin97] trabalham com uma rede completa, cujos canais são rotulados de acordo com a distância entre os processadores, baseando-se em um ciclo hamiltoniano previamente definido. O custo de

mensagens para os algoritmos de Loui *et al* e de Singh é o mesmo, $O(n \log n)$ bits. No entanto, a proposta de Singh diminui o número de passos necessários para se completar a eleição.

Ainda fazendo uso do conceito de senso de direção dois trabalhos abordando hipercubos foram desenvolvidos simultanea, mas independentemente ([FM93], posteriormente publicados em [FM96], e [Tel93]). Os dois algoritmos executam a eleição através de passos de disputas entre os sub-hipercubos, diferenciando-se na forma de obter a informação sobre os processadores que venceram nos passos anteriores. O custo total de mensagens dos dois algoritmos é o mesmo, $O(n \log n)$ bits, considerando que a identificação dos processadores tenha tamanho máximo igual a $\log n$.

A auto-estabilização é uma abordagem ainda muito estudada na área de algoritmos distribuídos. O desenvolvimento de um algoritmo auto-estabilizante é complexo, devido à difícil tarefa de demonstrar sua correção. Todos os algoritmos auto-estabilizantes descritos no presente trabalho [Hua93, ILS95, DIM97] utilizam a abstração de memória compartilhada e não fazem estudo de complexidade de mensagens, que na maioria dos casos seria muito alta. A grande preocupação é em relação ao tempo de estabilização do algoritmo. Os algoritmos de Huang e de Itkis *et al* utilizam anel uniforme unidirecional de tamanho primo. O fato do tamanho não ser um número composto é essencial para que a auto-estabilização ocorra. Em Huang é assegurado que o algoritmo estabiliza após cada processador passar por no máximo $O(n)$ estados. O algoritmo de Itkis *et al* possui uma outra característica importante além da auto-estabilização, sua memória utilizada por processador é de tamanho constante, sendo necessárias $O(n^2)$ trocas de estados dos processadores para que o sistema estabilize. Dolev *et al* propuseram um algoritmo para eleição auto-estabilizante em redes genéricas, cuja idéia básica é formar uma árvore geradora da rede e eleger a raiz da árvore. Usando a definição de *round* da seção 4.3, o tempo para estabilização do algoritmo é igual a $O(\Delta D \log n)$ *rounds*, onde Δ é o grau máximo de um nó da rede e D o diâmetro da rede. Outros artigos interessantes que tratam do problema de auto-estabilização são [BP89, Her90, AS96, GG96].

No capítulo 5 são apresentados algoritmos que utilizam probabilidades, seja para conseguir quebrar a simetria em redes anônimas e realizar uma eleição com possibilidade de erro, seja para proporcionar probabilidades iguais de vitória a todos os participantes da eleição, independente de seu valor de identificação. Dentre os algoritmos para redes anônimas descritos neste capítulo existem os que utilizam ou não o conhecimento do tamanho da rede e os síncronos ou assíncronos.

Os algoritmos probabilísticos para redes não anônimas apresentados possuem o interesse de garantir a todos os competidores a mesma probabilidade de vitória. São apresentadas algumas variações do algoritmo básico para o caso de alguns processadores possuírem probabilidades maiores de vencer, ou ser necessária a seleção de mais de um processador, ou ainda quando apenas parte dos processadores desejar eleger-se.

Na tabela 6.1 temos uma visão dos algoritmos apresentados neste trabalho, onde são mostrados a referência de cada algoritmo, o modelo de sistema utilizado e as complexidades de mensagens, tempo e espaço. Fica difícil fazer uma comparação visto que os algoritmos utilizam modelos diferentes.

O aspecto de tolerância a falhas, não abordado neste trabalho, é um problema que pode ser

tratado ortogonalmente, embora alguns autores tenham desenvolvido soluções que incorporam tolerância a falhas [AA87, Tau89, Sin96, KB96].

Com a pesquisa realizada pudemos ter conhecimento do grande número de trabalhos existentes na área de eleição de líder e de sua diversidade, possibilitando a aprendizagem de novas técnicas e abordagens empregadas para o desenvolvimento de algoritmos distribuídos.

Ref.	Modelo	Mensagens	Tempo	Espaço
[LeL77]	anel unidirecional com identificadores (Seção 2.1)	$O(n^2)$	$O(n)$	
[CR79]	anel unidirecional com identificadores (Seção 2.2)	$O(n^2)$	$O(n)$	
[Pet82]	anel unidirecional com identificadores (Seção 2.3)	$O(n \log n)$	$O(n \log n)$	
[GM82]	rede completa com identificadores (Seção 2.4)	$O(n^2)$	$O(1)$	
[LMW86]	rede completa com identificadores e senso de direção (Subseção 3.1.1)	$O(n)$	$O(n)$	
[Sin97]	rede completa com identificadores e senso de direção (Subseção 3.1.2)	$O(n)$	$O(\log n)$	
[FM93]	hipercubo com senso de direção (Subseção 3.2.1)	$O(n)$	$O(\log^3 n)$	
[Tel93]	hipercubo com senso de direção (Subseção 3.2.2)	$O(n)$	$O(\log^2 n)$	
[Hua93]	anel bidirecional de tamanho primo auto-estabilizante (Subseção 4.2.1)		$O(n)$	$O(\log n)$
[ILS95]	anel bidirecional de tamanho primo auto-estabilizante (Subseção 4.2.2)		$O(n^2)$	$O(1)$
[DIM97]	rede genérica auto-estabilizante (Seção 4.3)		$O(\Delta D \log n)$	$O(\log n)$
[IR90]	anel síncrono unidirecional sem ident. com n conhecido (Subseção 5.1.1)	$O(n)$	$O(n)$	
[IR90]	anel assíncrono unidirecional sem ident. com n conhecido (Subseção 5.1.1)	$O(n \log n)$	$O(n \log n)$	
[IR90]	anel síncrono unidirecional sem ident. com n desconhecido (Subseção 5.1.2)	$O(n)$	$O(N)$	
[IR90]	anel assíncrono unidirecional sem ident. com n conhecido (Subseção 5.1.2)	$O(n \log n)$	$O(n \log n)$	
[MA89]	rede genérica (Subseção 5.1.2)	$O(mr \log n \log r)$. $O(\log \log n + logr)$	$O(D)$	
[Ita90]	anel assíncrono unidirecional sem ident. com n desconhecido (Subseção 5.1.2)	$O(n^2 \log n)$		$O(1)$
[Cha90] e [VAD94]	rede completa com identificadores (Seção 5.2)	$O(n^2)$	$O(1)$	$O(n \log n)$

Tabela 6.1: Tabela com tabela com todos os algoritmos apresentados e suas complexidades.

Bibliografia

[AA87] H. H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *Lecture Notes in Computer Science*, 312:355–373, 1987. Proceedings of 2nd International Workshop on Distributed Algorithms.

[AG91] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM J. Comput.*, 20(2):376–394, 1991.

[AM94] Y. Afek and Y. Matias. Election in anonymous networks. *Information and Computation*, 113(2):312–330, 1994.

[Ang80] D. Angluin. Local and global properties in networks of processes. In *12th Annual ACM Sumposium on Theory of Computing*, pages 82–93, 1980.

[AS96] G. Antoniou and P. K. Srimani. A self-stabilizing leader election algorithm for tree protocol. *Journal of Parallel and Distributed Computing*, 34(2):227–232, 1996.

[Bar96] V. Barbosa. *An Introduction to Distributed Algorithms*, chapter 7, pages 183–200. The MIT Press, 1 edition, 1996.

[BP89] J. E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, April 1989.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*, chapter 10, pages 305–308. Addison-Wesley, 2 edition, 1994.

[Cha90] C. Chang. Bidding against competitors. *IEEE Transactions on Software Engineering*, 16(1):100–104, January 1990.

[CR79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of process. *Communications of the ACM*, 22(5):281–283, May 1979.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, November 1974.

[DIM97] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–439, April 1997.

[FM93] P. Flocchini and B. Mans. Optimal elections in labeled hypercubes. Technical Report TR-93-231, School of Computer Science, Carleton University, Ottawa, Canada, 1993.

[FM96] P. Flocchini and B. Mans. Optimal elections in labeled hypercubes. *Journal of Parallel and Distributed Computing*, 33:76–83, 1996.

[FMS] P. Flocchini, B. Mans, and N. Santoro. Sense of direction: Definitions, properties and classes. Networks, accepted October 1997.

[FMS96] P. Flocchini, B. Mans, and N. Santoro. On the impact of sense of direction on communication complexity. Technical Report TR-96-25, School of Computer Science, Carleton University, Ottawa, Canada, 1996.

[GG96] S. Ghosh and A. Gupta. An exercise in fault-containment: Self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, 1996.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[GM82] H. Garcia-Molina. Elections in distributed computing systems. *IEEE Transaction on Computers*, C-31(1):48–59, January 1982.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, June 1990.

[HS80] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, November 1980.

[Hua93] S. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, July 1993.

[ILS95] G. Itkis, C. Lin, and J. Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. *Lecture Notes in Computer Science*, 972:288–302, 1995. Proceedings of 9th International Workshop on Distributed Algorithms.

[IR90] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

[Ita90] A. Itai. On the computational power needed to elect a leader. *Lecture Notes in Computer Science*, 486:29–40, 1990. Proceedings of 4th International Workshop on Distributed Algorithms.

[KB96] J. L. Kim and G. G. Belford. A distributed election protocol for unreliable networks. *Journal of Parallel and Distributed Computing*, 35(1):35–42, 1996.

[KMZ84] E. Korach, S. Moran, and S. Zaks. Tight upper and lower bounds for some distributed algorithms for a complete network of processors, 1984.

[Lav95] C. Lavault. *Evaluation des algorithmes distribués.* HERMES, 1995.

[LeL77] G. LeLann. Distributed systems - toward a formal approach. *Proceedings of the IFIP Congress*, pages 155–160, 1977.

[LMW86] M. C. Loui, T. A. Matsushita, and D. B. West. Election in a complete network with sense of direction. *Information Processing Letters*, 22:185–187, 1986. Correção: Information Processing Letters, 28:327,1988.

[LR86] I. Lavalee and G. Roucariol. A fully distributed (minimal) spanning tree algorithm. *Information Processing Letters*, 23:55–62, 1986.

[Lyn96] N. Lynch. *Distributed Algorithms*, chapter 3,15. Morgan Kaufmann, 1 edition, 1996.

[MA89] Y. Matias and Y. Afek. Simple and efficient election algorithms for anonymous networks. *Lecture Notes in Computer Science*, 392:183–194, 1989. Proceedings of 3rd International Workshop on Distributed Algorithms.

[Pet82] G. L. Peterson. An o(n log n) unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, October 1982.

[PKR84] J. Pachl, E. Korach, and D. Rotem. Lower bounds for distributed maximum finding algorithms. *Journal of ACM*, 31:905–918, 1984.

[Ray88] M. Raynal. *Algorithms and Protocols*, chapter 2, pages 29–40. Jonh Wiley & Sons, 1988.

[SG94] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*, chapter 18, pages 595–600. Addison-Wesley, 4 edition, 1994.

[Sin96] G. Singh. Leader election in presence of link failures. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):231–236, March 1996.

[Sin97] G. Singh. Efficient leader election using sense of direction. *Distributed Computing*, 10(3):159–165, 1997.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*, chapter 11, pages 483–485. Prentice Hall, 1992.

[Tau89] G. Taubenfeld. Leader election in the presence of n-1 initial failures. *Information Processing Letters*, 33(1):25–28, October 1989.

[Tel93] G. Tel. Linear election for oriented hypercubes. Technical Report TR-RUU-CS-93-39, Department of Computer Science, Utrecht University, Utrecht, The Netherlands, 1993.

[Tel94] G. Tel. *Introduction to Distributed Algorithms*, chapter 2,7. Cambridge University Press, 1 edition, 1994.

[Tel95] G. Tel. Sense of direction in processor networks. *SOFSEM'95, Lecture Notes on Computer Science*, 1012, 1995.

[Tsi93] Y. H. Tsin. Incremental distributed asynchronous algorithm for minimum spanning trees. *Computer Networks and ISDN Systems*, 26:227–232, 1993.

[VAD94] F. Vernadat, P. Azema, and K. Drira. Distributed coin tossing, June 1994.