Eleição Assíncrona de Líder em Memória Compartilhada Dinâmica

Cátia Khouri[1,2], **Fabíola Greve**[2]

[1]DCET – U. Estadual do Sudoeste da Bahia (UESB), Vitória da Conquista, BA – Brasil

[2]D. de Ciência da Computação – U. Federal da Bahia (UFBA), Salvador, BA – Brasil

Abstract. *Cloud systems, SANs (storage area networks), etc. are inherently dynamic. The eventual leader detector is a powerful abstraction to ensure consistency and fault-tolerance in face of such dynamics. This paper presents a leader service, of class Ω, for the shared dynamic memory model, subject to process crashes. Differently from known solutions, it does not require the knowledge of the cardinality of the system membership, in such a way that nodes can leave and join the system as they wish.*

Resumo. *Sistemas distribuídos em nuvens, redes de área de armazenamento (SAN's), etc. são inerentemente dinâmicos. O detector de líder é uma abstração poderosa para garantir a consistência e tolerância a falhas em face de tal dinamismo. Este artigo apresenta um serviço de líder, de classe Ω, para o modelo de memória dinâmica compartilhada, sujeito a falhas de processos. Diferentemente das soluções conhecidas, ele não requer o conhecimento da cardinalidade dos participantes do sistema, de tal forma que nós podem sair e entrar no sistema livremente.*

1. Introdução

Sistemas distribuídos atuais desenvolvidos sobre redes de sensores, redes móveis *ad hoc*, grades oportunistas, computação em nuvem, redes de área de armazenamento (SAN's), redes P2P não-estruturadas, etc. possibilitam a seus participantes acessarem serviços e informações independentemente de sua localização ou mobilidade. Isso é possível através de um desenho de sistema altamente dinâmico que dispensa a existência de uma autoridade administrativa centralizada ou uma infraestrutura estática. Questões relacionadas ao desenho de soluções confiáveis que lidam com a natureza de alto dinamismo e auto-organização de tais redes são um campo muito ativo de pesquisa.

Por outro lado, no projeto de sistemas distribuídos confiáveis, camadas superiores de serviço (consenso, difusão atômica, replicação em máquinas de estados, etc.), usam um *serviço de eleição de líder* para prover os processos com a identidade de um líder, i.e., um processo correto no sistema. Tal serviço é tido como um oráculo distribuído que fornece a identidade de um processo considerado correto para todo nó que o invoca. Um oráculo da classe Ômega (Ω) pode fornecer informações equivocadas durante um período finito de tempo, mas a partir de um instante, todo processo correto recebe a identidade do mesmo processo correto. Diz-se então que todo processo no sistema confia naquele eleito como líder. A classe Ω reúne as condições minimais para se resolver o consenso [Chandra et al. 1996], um serviço fundamental que fatora a necessidade de acordo no sistema.

Este artigo propõe a implementação de um serviço de eleição de líder num sistema dinâmico assíncrono sujeito a falhas por parada (*crash*), em que os processos se comunicam através de uma memória constituída de registradores atômicos compartilhados.

O modelo de interação por memória compartilhada adequa-se bem a sistemas em que discos conectados em rede aceitam requisições de leitura e escrita diretamente de clientes. Esses discos constituem uma SAN (*storage area network*), que implementa uma abstração de memória compartilhada e tem se tornado cada vez mais frequente como mecanismo de tolerância a falhas. [Gafni and Lamport 2003] e [Guerraoui and Raynal 2007] apresentam algoritmos para tais sistemas baseados no detector de líder Ω. Adicionalmente, o desenvolvimento crescente de modernas arquiteturas multi-núcleo tem impulsionado estudos em sistemas de memória compartilhada assíncronos para os quais Ω tem sido aplicado como gerenciador de contenção, como em [Aguilera et al. 2003], [Fernández et al. 2010]. Apesar da importância de tais ambientes hoje, poucas são as propostas de modelos e protocolos para a implementação de Ω nessas circunstâncias.

Algumas abordagens para implementação de Ω no modelo de passagem de mensagens foram propostas. Inicialmente, elas consideravam um modelo com fortes requisitos de sincronia e confiabilidade dos canais [Chandra et al. 1996, Aguilera et al. 2001], mas, paulatinamente, foram sendo apresentadas soluções com requisitos cada vez mais fracos, considerando particularmente a existência de canais com requisitos temporais apenas para o líder [Aguilera et al. 2004, Malkhi et al. 2005, Hutle et al. 2009]. Entretanto, a totalidade de tais propostas são para o modelo clássico de sistemas distribuídos, cujo conjunto de processos é conhecido, além do número máximo f de falhas. Alguns poucos trabalhos existem para sistemas dinâmicos, cujo conjunto de processos é desconhecido e varia ao longo da execução; eles diferem no grau de conectividade e de conhecimento requerido do sistema [Jiménez et al. 2006, Fernández et al. 2006].

Para a comunicação por memória compartilhada, poucas são as propostas para Ω e a maioria delas considera sistemas estáticos [Guerraoui and Raynal 2006, Fernández et al. 2007, Fernández et al. 2010]. Apenas [Baldoni et al. 2009] incorpora alguns aspectos para tratar com o dinamismo do sistema, mas todas assumem canais com requisitos temporais. Uma estratégia, menos frequente, consiste em assumir um *padrão de mensagens* no sistema, ao invés de considerar limites para passos de processos ou transferência de mensagens. Protocolos baseados nesta abordagem não utilizam temporizadores e são por isso chamados *livres de tempo* (ou *time-free*). Nessa linha, alguns trabalhos surgiram para o modelo de passagem por mensagem [Mostefaoui et al. 2003, Arantes et al. 2013].

Até onde sabemos, até recentemente, não existia uma implementação de Ω para memória compartilhada assíncrona baseada em suposições livres de tempo. Em [Khouri and Greve 2014b], nós apresentamos uma primeira solução nesse sentido. Entretanto, o protocolo ali proposto não é eficiente, pois exige que todos os processos continuem a escrever na memória para sempre, mesmo após a eleição do líder. Os resultados ali apresentados servem como prova de conceito de que a eleição livre de tempo é possível.

Neste artigo, avançamos nessa compreensão do problema e apresentamos um protocolo Ω que, além de ser livre de tempo, é eficiente quanto às escritas. No caso, ele reune as condições necessárias para atingir escritas ótimas, que é quando, após a eleição,

somente o líder precisa continuar a escrever no seu registrador, enquanto os demais precisarão apenas ler. Isso ocorre sempre que os demais processos conseguem obter o último valor atualizado do registrador do líder. Ou seja, se todos os demais processos acessam a memória do líder somente depois que ele a atualizou. Com essa estratégia, tal como em [Aguilera et al. 2004, Malkhi et al. 2005, Hutle et al. 2009] que visavam optimalidade de sincronia, requerendo emissão de mensagens através de canais temporais apenas pelo líder, avançamos no estado da arte da implementação de Ω para memória compartilhada, requerendo que somente o líder continue a escrever na memória.

2. Modelo do Sistema

O sistema é composto por um conjunto infinito Π de processos que se comunicam através de uma memória compartilhada. Não fazemos suposições sobre a velocidade relativa dos processos, para dar passos ou acessar a memória, isto é, o sistema é assíncrono. Para facilitar a apresentação, assumimos a existência de um relógio global que não é acessível pelos processos, cuja faixa de ticks, $\mathcal{T}$, é o conjunto dos números naturais.

Consideramos o *modelo de chegada finita* [Aguilera 2004], isto é, o sistema tem um número infinito de processos que podem entrar e sair em instantes arbitrários, mas em cada execução r o número de participantes, n, é limitado, embora desconhecido. Os processos podem falhar parando arbitrariamente (*crashing*). Um processo *correto* comporta-se conforme a especificação, caso contrário, é *faltoso*. Cada processo p_i localiza-se num nó distinto e possui uma identidade distinta, i. O conjunto dessas identidades é totalmente ordenado. Denotamos um processo pelo seu nome: p_i, ou por sua identidade: i.

A memória compartilhada distribuída é uma abstração construída no topo de um sistema de passagem de mensagens que permite aos processos comunicarem-se invocando operações sobre objetos compartilhados. No nosso modelo, a memória consiste de arranjos de registradores atômicos multivalorados do tipo 1-*escritor-Múltiplos-leitores* ($1WMR$) que se comportam corretamente [Lamport 1986]. Um registrador é um tipo de objeto compartilhado que armazena um valor inteiro e provê duas operações básicas: $read$, que retorna o valor armazenado; e $write$, que escreve (armazena) um valor no registrador. Um registrador $R[i]$ pertence ao processo p_i, que é o seu único escritor, mas pode ser acessado por qualquer p_j para leitura.

Comportamento correto de um registrador significa que ele sempre pode executar uma leitura ou escrita e nunca corrompe o seu valor. Uma leitura sobre um *registrador atômico* que não é concorrente a uma escrita, obtém o último valor escrito no registrador. Quando uma leitura é concorrente a uma ou mais escritas sobre um registrador atômico pode retornar o valor antigo (escrito pela última operação de escrita não-concorrente à leitura), ou o valor sendo escrito por uma das escritas concorrentes [Lamport 1986].

Assumimos que a implementação da memória provê as operações $join()$ e $leave()$ que são invocadas pelos processos para entrar e sair, respectivamente, da computação. Um processo p_i que deseja entrar não têm conhecimento sobre o conjunto de participantes nem o estado dos objetos compartilhados. Assim, p_i invoca a função $join()$ que lhe provê uma identidade e o estado consistente do sistema; e informa aos demais nós sobre sua chegada. Dizemos que p_i está ativo a partir do retorno de $join()$ até que falhe ou deixe o sistema. Se um processo sai e volta para a computação, ele é considerado um novo processo e portanto terá uma nova identidade. Definimos assim o conjunto de processos ativos:

Definição 1 (ACTIVE) $ACTIVE(t) = \{p_a : p_a \textit{ está ativo no instante } t\}$.

A despeito da entrada e saída de processos, a existência de um conjunto de processos estáveis num sistema dinâmico é necessária para que se possa realizar alguma computação útil. A noção de *estável* refere-se a um processo que entra no sistema e permanece ativo durante toda computação. O parâmetro α representa o limite inferior para a cardinalidade do conjunto de processos estáveis.

Definição 2 (STABLE) $STABLE = \{p_s : \exists t \in \mathcal{T}, \forall t' \geq t : p_s \in ATIVO(t')\}$

Para sistemas distribuídos clássicos de passagem de mensagem, o número de processos no sistema, n, e o limite superior para o número de falhas, f, desempenham papel fundamental no projeto de algoritmos, conquanto juntos, eles representam um limite inferior sobre o número de processos corretos para que o algoritmo convirja. Em um sistema dinâmico, onde nem n nem f são conhecidos, uma estratégia adotada é abstrair o par (n, f) através de um parâmetro α, que representa esse número mínimo de processos estáveis no sistema. É bem sabido que com passagem de mensagens, o limite para o número de processos faltosos é $n/2$, enquanto que com memória compartilhada, esse limite sobe para $n - 1$ [Lo and Hadzilacos 1994]. Considerando a possibilidade de $f \leq (n - 1)$ processos falharem, temos $\alpha \geq (n - (n - 1))$, então $\alpha \geq 1$. De modo semelhante, no modelo proposto, assumimos a existência de um parâmetro α, que define o número mínimo de processos $p_s \in STABLE$. Portanto, uma condição de progresso provida pelo modelo do sistema é dada pela Propriedade 1.

Propriedade 1 (Limite Inferior de $STABLE$) $|STABLE| \geq \alpha$.

2.1. Especificação de Ω

O uso de abstrações como detectores de falhas ou líderes possibilita a solução de certos problemas insolúveis em sistemas assíncronos sujeitos a falhas [Fischer et al. 1985]. Esses dispositivos funcionam como oráculos distribuídos que fornecem "dicas" sobre falhas de processos (ou líderes). Um serviço *detector de líder após um tempo* da classe Ω provê os processos com uma função *Leader()* que, quando invocada por um processo p_i, fornece a identidade de um processo p_j que o detector considera correto naquele instante. Após um tempo, essa identidade é única para todos os processos que invocam a função. Num contexto de sistema dinâmico, um detector Ω satisfaz à seguinte propriedade:

Liderança após um tempo (*eventual leadership*): existe um instante após o qual qualquer invocação de $Leader()$ por qualquer processo $p_i \in STABLE$ retorna a mesma identidade de processo $p_l \in STABLE$.

3. Eleição de Líder num Sistema Dinâmico de Memória Compartilhada

Para implementar Ω num sistema puramente assíncrono, é preciso enriquecer o sistema com suposições adicionais de sincronia que permitam a convergência. Em geral, essas suposições envolvem aspectos temporais, de modo que os protocolos utilizam temporizadores. Ao invés disso, introduzimos uma propriedade comportamental que reflete um padrão relativo aos acessos feitos à memória compartilhada.

3.1. Propriedade do Sistema

Considere um sistema em que cada processo ativo p_i escreve/atualiza periodicamente seu registrador $R[i]$ enquanto executa uma rodada de operações de leitura sobre os registradores $R[j]$ de p_j, a fim de obter os (novos) valores escritos. A rodada termina quando p_i tiver obtido um número "suficiente" de valores atualizados e, na sequência, p_i inicia uma nova rodada. Seja $U_i^{r_i}$ o conjunto dos primeiros α processos p_j cujos registradores foram encontrados atualizados por p_i na rodada r_i. A Propriedade 2 impõe a existência de um processo p_l que a partir de um instante t, está presente em $U_i^{r_i}, \forall r_i$, p_i é estável.

Propriedade 2 (Padrão de Acesso à Memória – PAM) *Existe um instante t e um processo $p_l \in STABLE$ tais que, $\forall p_i \in STABLE$, $\forall r_i$ iniciada após t : $p_l \in U_i^{r_i}$.*

Em última análise, a Propriedade 2 reflete alguma regularidade no comportamento do sistema, mesmo que ele se comporte de maneira assíncrona por um período arbitrário de tempo. É como se a partir de t, a comunicação entre cada processo p_i e p_l estivesse entre as (α) mais rápidas quando se considera a comunicação entre p_i e cada p_j.

3.2. Visão Geral do Algoritmo de Eleição de Líder

O Algoritmo é executado em paralelo por todos os processos ativos. Com o intuito de informar aos demais processos que está ativo, um processo p_x que se considera líder mantém-se incrementando um contador num registrador compartilhado $Alive[x]$. Por outro lado, periodicamente, cada p_i que considera p_x como líder, consulta o valor em $Alive[x]$ para saber se p_x progrediu. Se p_i não obtém um valor atualizado, antes de suspeitar de p_x, inicia uma rodada de leituras sobre cada $Alive[j]$ e espera encontrar pelo menos α registradores atualizados. Se $Alive[x]$ está entre os α primeiros encontrados, não é considerado suspeito, caso contrário, p_i suspeita de p_x e lhe aplica uma punição. Para esse fim, p_i mantém um arranjo de registradores – $Punishments$. Sempre que p_i suspeita de p_j, incrementa $Punishments[i][j]$. Como os processos compartilham a mesma visão de $Punishments$, p_i escolhe como líder o processo p_j para o qual $soma_i[j] = \sum_{\forall k} Punishments[k][j]$ é mínima. Se houver mais de uma soma com valor igual ao mínimo, o desempate será definido pela ordem lexicográfica, considerando as identidades dos processos. Dado o arranjo $soma_i$, a função $MinLex(soma_i)$ retorna x tal que:
$\forall x, y, (soma_i[x] = soma_i[y] \wedge x < y) \vee (soma_i[x] < soma_i[y])$.

No algoritmo são utilizados os seguintes registradores compartilhados:

- $Alive[i]$ – contador que indica o progresso de p_i; inicializado com 0.
- $Punishments[i][j]$ – contador de p_i para o número de punições aplicadas por p_i a p_j. No início, $Punishments[i][j] = 0, \forall i, j$. Quando um processo p_r entra depois, p_i inicializa $Punishments[i][r]$ com $Punishments[i][leader_i] + 1$.

Um processo p_i também utiliza as seguintes variáveis locais:

- $leader_i$ e ld_i – identidade do líder corrente em TASK 1 e TASK 2.
- $alive_i$ – controla o progresso de p_i cujo valor é escrito em $Alive[i]$. Inicializado com 0.
- $last\_leader_i$ e $last\_ld_i$ – identidade do líder da iteração anterior em TASK 1 e TASK 2.
- $last\_alive\_leader_i$ – último valor lido em $Alive[leader_i]$.
- $alive_i^j$ – variável auxiliar usada para leitura de $Alive[j]$.
- $last\_alive_i$ – vetor dinâmico que armazena o último valor lido em $Alive[1..]$. Cada entrada é inicializada com 0.

Algorithm 1 Leader_Election() - Part 1

```
    INITIALIZATION
 1: join();
 2: start TASK 0, TASK 1, TASK 2;
    TASK 0: INIT-REGS
    UPON REQUEST join() FROM p_j
 3: write(Punishments[i][j], Punishments[i][Leader()] + 1);
    TASK 1: ALIVENESS
 4: loop
 5:     leader_i ← Leader();
 6:     if (i = leader_i) then
 7:         alive_i ← alive_i + 1;
 8:         write (Alive[i], alive_i);
 9:         if (last_leader_i ≠ leader_i) then;
10:             last_leader_i ← leader_i;
11:     else
12:         read(Alive[leader_i], alive_leader_i);
13:         if (last_leader_i = leader_i) then
14:             if (last_alive_leader_i = alive_leader_i) then
15:                 alive_i ← alive_i + 1;
16:                 write (Alive[i], alive_i);
17:         else
18:             last_leader_i ← leader_i;
19:             last_alive_leader_i ← alive_leader_i;
```

- $soma_i$ – vetor dinâmico que armazena a soma dos contadores de punições de cada p_j.
- $updated_i$ – conjunto para controle dos processos que têm seu progresso verificado.
- pun_i – vetor dinâmico que armazena a quantidade de punições aplicadas por p_i a cada p_j. Cada entrada é inicializada com 0.

Notação: Dizemos que "p_i considera p_x como líder", se $leader_i = x$ (em TASK 1) ou $ld_i = x$ (em TASK 2). Dizemos que "p_x é demovido da posição de líder", se num instante t, $leader_i = x$ ($ld_i = x$) e na próxima atribuição de valor, $leader_i$ (ld_i) recebe $y \neq x$. Dizemos que "p_i pune p_j" quando p_i incrementa seu registrador $Punishments[i][j]$. "Atualizar um registrador", significa escrever no registrador seu valor corrente incrementado de 1. Um registrador $Alive[j]$ é considerado "atualizado" por p_i se ele obtém numa leitura um valor maior que o obtido na leitura imediatamente anterior.

3.3. Descrição do Algoritmo

Para um processo p_i entrar no sistema, invoca uma operação $join()$ que lhe provê: uma identidade; um estado consistente dos registradores; inicializa seus registradores compartilhados; e aloca espaço para seus arranjos dinâmicos locais. Os demais processos p_j, ao receberem uma requisição de $join()$, criam seus registradores $Punishments[j][i]$ devidamente inicializados e alocam espaço nos arranjos dinâmicos locais para acomodar p_i. Apenas após o retorno de $join()$, um processo p_i estará apto a participar da computação. Então p_i inicia três tarefas em paralelo.

Algorithm 2 Leader_Election() - Part 2

TASK 2: PUNISHMENT

20: **loop**
21: $ld_i \leftarrow Leader()$;
22: **if** $(ld_i \neq i)$ **then**
23: $updated_i \leftarrow \{i\}$;
24: **while** $((ld_i \notin updated_i) \wedge (|updated_i| < \alpha))$ **do**
25: **read** $(Alive[ld_i], alive\_ld_i)$;
26: **if** $(alive\_ld_i \neq last\_alive_i[ld_i])$ **then**
27: $last\_alive_i[ld_i] \leftarrow alive\_ld_i$;
28: $updated_i \leftarrow updated_i \cup ld_i$;
29: **else if** $(ld_i \notin updated_i)$ **then**
30: **for all** (j) **do**
31: **if** $(j \notin updated_i)$ **then**
32: **read** $(Alive[j], alive_i^j)$;
33: **if** $(alive_i^j \neq last\_alive_i[j])$ **then**
34: $last\_alive_i[j] \leftarrow alive_i^j$;
35: $updated_i \leftarrow updated_i \cup j$;
36: **if** $((|\ updated_i\ | < \alpha) \wedge (ld_i \notin updated_i))$ **then**
37: **if** $(leader_i \neq ld_i)$ **then** $ld_i \leftarrow leader_i$;
38: **if** $(ld_i \notin updated_i)$ **then**
39: **for** $(j \notin updated_i)$ **do**
40: $pun_i[j] \leftarrow pun_i[j] + 1$;
41: **write**$(Punishments[i][j], pun_i[j])$;

$Leader()$

42: **for all** (j) **do**
43: $soma_i[j] \leftarrow 0$;
44: **for all** (k) **do**
45: **read**$(Punishments[k][j], pun1_i)$;
46: $soma_i[j] \leftarrow soma_i[j] + pun1_i$;
47: $leader_i \leftarrow MinLex(soma_i[j])$;
48: **return** $leader_i$;

TASK 0: INIT-REGS – Ao perceber uma requisição de $join()$ de algum p_j, p_i inicializa o registrador $Punishments[i][j]$ com $Punishments[i][Leader()] + 1$ (linha 3).

TASK 1: ALIVENESS – Nesta tarefa, quando um processo p_i se considera líder, mantém-se atualizando $Alive[i]$, enquanto cada p_j verifica se $Alive[i]$ está atualizado. Consiste de um laço infinito no qual, inicialmente, cada processo p_i invoca a função $Leader()$ para definir $leader_i$ (linha 5). Para fazer certas verificações na iteração seguinte, p_i mantém a identidade do líder e de seu respectivo registrador $Alive$ em variáveis locais ($last\_leader_i$ – linhas 10 e 18) e ($last\_alive\_leader_i$ – linha 19). Enquanto p_i considera-se líder ($leader_i = i$), mantém-se atualizando $Alive[i]$ (linhas 6-8).

Um p_i que não se considera líder ($leader_i \neq i$),mantém-se verificando se $leader_i$ está vivo. Para isso, p_i lê $Alive[leader_i]$ (linha 12) e verifica se o líder corrente é o mesmo da iteração anterior (linha 13). Se isso ocorre e $Alive[leader_i]$ não foi atualizado (linha

14), p_i atualiza seu $Alive[i]$ (linhas 15-16). Senão, p_i apenas guarda a identidade do líder e o valor de seu registrador (linhas 18 e 19). Isso significa que é possível que apenas o líder eleito permaneça escrevendo em $Alive$ pra sempre.

TASK 2: PUNISHMENT – Aqui também são utilizadas variáveis locais, $last\_alive_i[j]$, para guardar o valor lido em $Alive[j]$ e possibilitar a verificação do progresso de p_j (linhas 27, 34). O objetivo da tarefa é punir processos suspeitos, mas isso só ocorre se $leader_i$ é um dos suspeitos. Então, no início do laço, p_i define um líder, ld_i, invocando $Leader()$ (linha 21). Nenhum processo suspeita de si mesmo. Então, enquanto p_i se considera líder, apenas invoca $Leader()$ (linhas 21-22). Já um processo $p_i \neq ld_i$, inicializa o conjunto $updated_i$ com sua própria identidade (linha 23) e entra num laço **while** (linha 24) onde permanece até que uma das condições seja satisfeita – *Condição 1*: $ld_i \in updated_i$, o que significa que $Alive[ld_i]$ foi atualizado; ou *Condição 2*: $\mid updated_i \mid \geq \alpha$, que implica que pelo menos α processos p_j atualizaram $Alive[j]$. Como o algoritmo foi desenhado para sistemas assíncronos estendidos com as Propriedades 1 e 2, essas condições garantem que, ao cabo de um tempo, algum processo p_l sempre estará em $updated_i, \forall i$.

No **while**, p_i lê $Alive[ld_i]$ (linha 25). Se obtém um valor atualizado, guarda-o em $last\_alive_i[ld_i]$ e inclui ld_i em $updated_i$ (linhas 26-28). Dessa forma, a *Condição 1* é imediatamente satisfeita. Então, p_i sai do laço, avalia o predicado da linha 38: *falso* e não pune qualquer p_j. Se, no entanto, p_i não obtém um valor atualizado, entra num laço **for** (linhas 30-35) onde lê cada $Alive[j]$ e, caso obtenha um valor atualizado, inclui j em $updated_i$. Após o **for**, se a *Condição 1* é satisfeita, p_i sai do **while** e vai para a próxima iteração do **loop**. Se a *Condição 2* é satisfeita e $ld_i \notin updated_i$, p_i pune cada $p_j \notin updated_i$ e vai para a próxima iteração do **loop** (linhas 38-41). Para evitar que um processo permaneça bloqueado no **while**, após cada iteração do **for**, p_i verifica se o líder que ele obteve na última invocação a $Leader()$ em TASK 1 – $leader_i$, é o mesmo sendo considerado em TASK 1 – ld_i. Se não for, ele altera ld_i para receber $leader_i$ (linhas 36-**??**). Quando a Propriedade 2 se verifica, algum p_l eleito como líder por todo p_i não será mais punido, portanto nenhuma escrita mais será efetuada sobre $Punishments$.

A função Leader() – Retorna uma identidade de processo que, do ponto de vista de p_i, é correto e considerado líder por ele. Para essa escolha, a soma das punições atribuídas a cada p_j é acumulada em $soma_i[j]$ (linhas 42-46). É escolhido como líder o processo com menor soma, utilizando a identidade como desempate (linhas 47-48).

4. Corretude do Algoritmo

Esta seção apresenta um resumo da prova de que os processos estáveis elegem um único processo estável – o líder, ao cabo de um tempo. Uma versão completa da prova pode ser obtida em [Khouri and Greve 2014a], que devido à restrições de espaço não pode ser aqui detalhada.

Lema 1 *Existe um instante t e um processo $p_l \in STABLE$ tais que após t, p_l não será jamais punido por qualquer processo p_i.*

Prova. As punições ocorrem na TASK 2 (linhas 38-41). Se $p_i \in STABLE$, de acordo com a Propriedade 2, existem um processo p_l e um instante t', tais que em toda rodada de leituras sobre os registradores $Alive$, iniciada por p_i após t', resultará na inclusão de p_l em $updated_i$ (linhas 25-28 ou 30-35), antes que p_i saia do **while** da linha 24. Logo, a partir de um instante $t > t'$, p_l não será punido por qualquer p_i. $_{Lema1}\square$

Lema 2 *Nenhum processo p_i fica permanentemente bloqueado executando o Algoritmo Leader_Election().*

Prova. O único trecho do algoritmo em que um processo pode ficar bloqueado é o laço **while** da TASK 2 (linhas 24-**??**), o qual só é executado por processos que não se consideram líderes (linha 22). Para mostrar que um processo não fica bloqueado no laço, devemos mostrar que pelo menos uma das condições é satisfeita: *Condição 1*: $ld_i \in updated_i$; ou *Condição 2*: $\mid updated_i \mid \geq \alpha$ (linha 24). Considere os seguintes casos possíveis em que um processo estável p_i que considera p_x como líder executa o **while** da TASK 2.

Caso 1: p_x é estável e considera-se líder. Logo, atualiza $Alive[x]$ (linha 8). Ao cabo de um tempo, p_i obtém $Alive[x]$ atualizado (linha 25 ou 32) e a *Condição 1* é satisfeita.

Caso 2: p_x é faltoso e falha em um instante t quando pelo menos α processos estáveis p_j o consideram como líder na TASK 1 (linha 5). Então, uma vez que nenhum p_j obtém $Alive[x]$ atualizado, p_j atualiza $Alive[j]$ (linhas 12-16). Na TASK 2, p_i inclui cada p_j em $updated_i$ (linhas 30-35) e após um tempo, a *Condição 2* é satisfeita.

Caso 3: p_x é faltoso e falha no instante t, quando menos de α processos estáveis p_j o consideram como líder na TASK 1 (linha 5). Como p_i não inclui p_x em $updated_i$ (linhas 28 ou 35), a *Condição 1* não é satisfeita. Então, devemos mostrar que p_i obtém α registradores $Alive$ atualizados. Uma vez que $\mid STABLE \mid \geq \alpha$ (Propriedade 1), se menos de α processos estáveis p_j consideram p_x como líder, sabemos que existe algum processo p_k tal que $leader_k = y, y \neq x$. Considere o seguinte cenário, sem perda de generalidade. (I) No instante t, $\exists p_j : leader_j = x$ e $\exists p_k : leader_k = y$. (II) No intervalo de tempo $I = [t', t''], t' < t < t$“, as únicas operações de escrita sobre $Punishments$ em andamento, são operações sobre $Punishments[-][x]$ e $Punishments[-][y]$. (III) Existem operações de leitura sobre esses mesmos registradores executadas por p_j e p_k concorrentes às escritas (linhas 42-46). Daí temos que, (A) após um tempo, se nenhuma nova escrita é requisitada sobre esses registradores, todo p_h que invoca $Leader()$ na TASK 1 (linha 5), retorna z. Se z é correto, atualiza $Alive[z]$ (A1); Senão, todo p_j correto atualiza $Alive[j]$ (A2) (linhas 12-16). Em TASK 2 p_i obtém essas atualizações e a *Condição 1* ou *Condição 2* é satisfeita (De (A1) ou (A2)). Por outro lado, (B) para que novas punições ocorram, também é preciso que pelo menos α processos p_h suspeitem de p_z e atualizem $Alive[h]$ na TASK 1 (linhas 12-16). Assim, p_i obtém α $Alive[h]$ atualizados, inclui cada p_h em $updated_i$, e a *Condição 2* é satisfeita.

Caso 4: p_x é estável, mas não se considera líder. Então, p_x não atualiza $Alive[x]$ na linha 8 mas pode ser que o faça na linha 16, se suspeita de seu líder. Se isso ocorre, a *Condição 1* é satisfeita. Caso contrário, se pelo menos α p_j consideram p_x como líder, tal como nos *Casos 2* e *3*, a *Condição 2* é satisfeita. Senão, temos os vários desdobramentos do *Caso 3* e, ao cabo de um tempo, uma das duas condições é satisfeita. $_{Lema2}\square$

Lema 3 *Existe um instante t e um processo $p_l \in STABLE$ tais que $\forall t' \geq t$, $\forall i$, $leader_i = ld_i = p_l$.*

Prova. Em um instante t, p_i define um processo p_x como líder, se ao invocar $Leader()$, obtém um estado de $Punishments$ tal que $MinLex(\sum_{\forall k} Punishments[k][j]) = x$.

Pelo Lema 1, algum processo p_l (que satisfaz a Propriedade 2) não é mais punido a partir de um instante t. Isto é, seus contadores de punições permanecem inalterados, enquanto que os demais podem crescer. Assim, após um tempo, p_l terá o menor nível de punições e será eleito líder. Como não é mais punido, também não é demovido. $_{Lema3}\square$

Lema 4 *Se um processo p_x é considerado líder por algum processo p_i e falha no instante t, então, existe um instante $t' > t$, em que p_x é demovido da posição de líder, e a partir de um instante $t'' \geq t'$, p_x não será considerado líder por qualquer processo.*

Prova. Temos a considerar os seguintes casos.

Caso 1: No instante t, pelo menos α p_j consideram p_x como líder na TASK 1, quando p_x falha. A prova segue dos argumentos de prova do *Caso 2* – Lema 2.

Caso 2: Menos de α p_j consideram p_x como líder na TASK 1 (linha 5), quando p_x falha, no instante t. A prova segue da prova do *Caso 3* – Lema 2.

Teorema 1 *Existe um instante após o qual algum $p_l \in STABLE$ será eleito permanentemente líder.*

Prova. Segue direto dos Lemas 3, 1, 2 e 4. $Teorema1 \square$

5. Complexidade

O algoritmo usa n registradores $Alive$ e n^2 registradores $Punishments$, isto é, $O(n^2)$ registradores $1WMR$, onde n é o número de processos que entra no sistema. Uma vez que o número de rodadas não é limitado, o tamanho dos registradores $Alive$ é ilimitado também. No entanto, os registradores $Punishments$ são limitados, já que a partir do instante em que PAM é satisfeita, nenhum processo mais é punido.

Com relação à complexidade de passo, temos que, inicialmente, um processo p_i invoca a função $Leader()$ para definir um líder. Nesta função, nenhuma escrita é executada, e são realizadas $[n(r)]^2$ operações de leitura sobre $Punishments$, onde $n(r)$ é o número de processos que já entraram até a rodada r. Em TASK 0, p_i executa 1 escrita cada vez que um novo processo p_r entra no sistema. Nenhuma leitura é feita nesta tarefa. Em TASK 1, p_i executa n^2 operações de leitura (em $Leader()$) e então: (1) se $leader_i = i$, p_i executa 1 escrita em $Alive[i]$; (2) se $leader_i = j \neq i$, p_i executa 1 leitura em $Alive[j]$ e pode ter que executar 1 escrita ou não. Isto é, p_i executa, no máximo, 1 escrita.

Em TASK 2, p_i executa n^2 operações de leitura (em $Leader()$), e então: (1) se $leader_i = i$, p_i não executa qualquer acesso à memória compartilhada; (2) se $leader_i = j$, p_i executa (a) 1 leitura em $Alive[j]$ e nenhuma escrita, caso obtenha $Alive[j]$ atualizado nesta leitura; ou (b) pelo menos $n(r)$ leituras sobre $Alive[-]$, se obtém $Alive[j]$ atualizado em alguma dessas leituras, e nenhuma escrita; ou (c) no pior caso, p_i suspeita de p_j e executa, pelo menos $n(r)$ leituras sobre $Alive[-]$ e tantas punições quantos forem os registradores encontrados desatualizados, isto é, até $n(r) - \alpha$ escritas em $Punishments$.

Portanto, a partir do instante em que a propriedade 2 é satisfeita, e considerando que a chegada de processos cessou, em uma rodada, um processo p_i executa, no pior caso, $O(n^2)$ leituras e 1 escrita. Focando num sistema onde leituras são locais [Baldoni et al. 2009] (ou mesmo mais baratas que escritas), o custo real do algoritmo deve-se às operações de escrita. Por esse motivo, a abordagem utilizada no projeto do algoritmo foi a redução das escritas. Considerando execuções *bem-comportadas*, nas quais o registrador do líder é sempre encontrado atualizado, apenas o líder eleito permanece escrevendo para sempre na memória compartilhada. O mesmo ocorre no caso especial em que $\alpha = 1$. Nessas situações, o comportamento do algoritmo é ótimo, já que em qualquer algoritmo de Ω para o modelo proposto, pelo menos um processo, o líder eleito, p_l, precisa informar seu progresso pra sempre [Fernández et al. 2010].

6. Trabalhos Relacionados

A Tabela 1 resume as principais abordagens para eleição eventual de líder e implementação de Ω em sistemas assíncronos propostas na literatura. Podemos distinguir duas grandes categorias: (i) os que estendem o sistema assíncrono com requisitos temporais (a grande maioria) e são considerados (*timer-based*) e (ii) os que não usam o tempo, mas padrão de troca de mensagens ou de acesso à memória (*message-pattern* ou *time-free*). Como a Tabela 1 mostra, a grande maioria considera o modelo de passagem de mensagens e poucos são para o de memória compartilhada.

Abordagem	Comunicação	Π	Conectividade	Propriedade do Modelo
[Chandra et al. 1996]	pass. de mensagens	conhecido	grafo completo	sistema parcialmente síncrono
$\exists p$ *eventually accessible* [Aguilera et al. 2001]	pass. de mensagens	conhecido	grafo completo	$\exists p$: todos os canais bidirecionais de/para p são *eventually timely*
$\exists p$ que é $\diamondsuit f$-*source* [Aguilera et al. 2004]	pass. de mensagens	conhecido	grafo completo	$\exists p$: f canais de saída de p são *eventually timely*
$\exists p$ que é $\diamondsuit f$-*accessible* [Malkhi et al. 2005]	pass. de mensagens	conhecido	grafo completo	$\exists p, Q$: *round-trip* de p para $\forall q \in Q$, é *eventually timely*
$\exists p$ que é *eventually moving* f-*source* [Hutle et al. 2009]	pass. de mensagens	conhecido	grafo completo	$\exists p, Q$: canal de saída de p para $\forall q \in Q$, é *eventually timely*
[Jiménez et al. 2006]	pass. de mensagens	desconhecido dinâmico	nós corretos conectados	$\exists p$ correto: $\forall q$ correto, pode ser atingido de p por canal *eventually timely*
[Fernández et al. 2006]	pass. de mensagens	desconhecido dinâmico	nós corretos conectados	$\exists p$ correto e $>(n$-$f)$-$\alpha+1$ processos corretos alcançáveis de p por canais *eventually timely*
[Guerraoui and Raynal 2006]	memória compartilhada	conhecido	-	sistema síncrono
temporizadores "bem comportados" [Fernández et al. 2010]	memória compartilhada	conhecido	-	$\exists p$: cujos acessos à memória compartilhada são *eventually timely*
mecanismo *query-response* (*time-free*) [Mostefaoui et al. 2003]	pass. de mensagens	conhecido	grafo completo	$\exists t, p, Q$: a partir de t, $q \in Q$ recebe uma resposta de p à suas *queries* entre as $(n-f)$ primeiras.
mecanismo *query-response* (*time-free*) [Arantes et al. 2013]	pass. de mensagens	desconhecido dinâmico	grafo completo	$\exists t, p, Q$: a partir de t, $q \in Q$ recebe uma resposta de p à suas *queries* entre as α primeiras.
padrão de acesso à memória [Khouri and Greve 2014b]	memória compartilhada	desconhecido dinâmico	-	$\exists t, p_l$: a partir de t, $Alive[l]$ está entre os α primeiros registradores atualizados

Tabela 1. Abordagens para Implementação de Ω

6.1. Soluções Clássicas para Modelo de Sistemas Estáticos

A busca por requisitos mínimos relativos ao grau de sincronia e confiabilidade dos canais e seu grau de conectividade foram fruto de estudo de diversos trabalhos para o modelo de passagem de mensagem. As 7 entradas iniciais da Tabela 1 têm tal abordagem. Os primeiros estudos de detectores de líder consideravam uma rede de comunicação completamente conectada com todos os canais confiáveis e síncronos a partir de um determinado instante [Chandra et al. 1996]. Já [Aguilera et al. 2001] considera que existe pelo menos um processo correto p e um instante t, tais que a partir de t, todos os canais bidirecionais entre p e todos os outros processos são síncronos (*eventually timely*). [Aguilera et al. 2004] mostram que Ω pode ser implementado em sistemas com falhas por parada, cujos canais sejam *fair-lossy* (permitem perdas equitáveis), desde que pelo menos um processo correto p possua f canais de saída que sejam *eventually timely* (p é dito ser um $\diamondsuit f\text{-}source$).

Malkhi et al. (2005) usam uma estratégia diferente: a noção de *eventually* f-*acessibility*. Supõe-se um instante t, um processo p e um conjunto Q (variável com o

tempo) de f processos, tais que, $\forall t' \geq t$, p troca mensagens com cada q dentro do limite de tempo δ (relativo ao *round-trip* do canal), isto é, p é $\diamondsuit f$*-accessible*. não é fixo, isto é, pode variar com o tempo. Uma suposição ainda mais fraca é a proposta em [Hutle et al. 2009]: existe um processo correto p, a saber, um *eventually moving f-source*, tal que, após um tempo, cada mensagem que ele envia é recebida em tempo (*timely*) por um conjunto Q de f processos que pode ser diferente em instantes distintos.

Para memória compartilhada, há poucas propostas para Ω e a maioria delas considera sistemas estáticos. [Guerraoui and Raynal 2006] implementam um protocolo com registradores regulares mas diferente do nosso trabalho, consideram um sistema estático, síncrono após um tempo. [Fernández et al. 2007], [Fernández et al. 2010], consideram um sistema assíncrono com registradores atômicos compartilhados; mas, ao contrário da nossa proposta, supõem um ambiente estático, além de canais síncronos.

6.2. Soluções Recentes para Modelo de Sistemas Dinâmicos

Em sistemas dinâmicos, há uma imprevisibilidade sobre o conjunto de participantes da computação, bem como sobre os canais de comunicação. [Jiménez et al. 2006] propõem um protocolo para ambiente dinâmico com troca de mensagens, supondo canais *eventually timely*. Já [Fernández et al. 2006] apresentam dois protocolos. No primeiro eles consideram que os processos não conhecem o número de participantes n nem o limite para o número de falhas f. Como na nossa abordagem, cada processo só conhece sua própria identidade e um limite inferior α sobre o número de processos corretos. Diferentemente da nossa proposta, entretanto, os processos se comunicam por troca de mensagens e o algoritmo, baseado em *timeouts*, exige a existência de alguns canais *eventually timely*.

Em um sistema de memória compartilhada cujos participantes são desconhecidos, um processo que chega, naturalmente, precisa inteirar-se do estado do sistema antes de participar efetivamente. [Baldoni et al. 2009] propõem uma abstração de registrador regular compartilhado no topo de um sistema dinâmico de passagem de mensagens. Quando um processo p_i deseja entrar, invoca uma operação $join()$, que lhe permite obter um estado consistente do registrador. De modo semelhante, o nosso protocolo admite o uso de uma operação $join()$ que se encarrega de criar registradores devidamente inicializados.

Abordagens Livres de Tempos Baseadas em Padrão de Mensagens. [Mostefaoui et al. 2003, Mostefaoui et al. 2006] introduziram uma abordagem baseada num mecanismo *query-response* que assume que a resposta de algum p_i às últimas requisições de p_j está sempre entre as primeiras $n - f$ respostas recebidas por cada p_j. A intuição por trás da suposição é a de que, mesmo que o sistema não exiba propriedades síncronas, ele pode apresentar alguma regularidade de comportamento que se traduza numa *sincronia lógica* capaz de contornar a impossibilidade de conversão do algoritmo.

A partir de uma suposição semelhante, [Arantes et al. 2013] propõem um modelo para implementação de Ω num sistema dinâmico de passagem de mensagens que considera a mobilidade dos nós. Supondo um padrão de mensagens adaptado a sistemas dinâmicos, segundo o qual os processos envolvidos pertencem a um conjunto de nós *estáveis*, eles utilizam o limite inferior α sobre o número de processos estáveis para controlar o número de respostas consideradas entre as primeiras. Essa constante α abstrai todos os pares (n, f) tradicionalmente consideradas, tal que $n - f \geq \alpha$. Dessa forma, um protocolo baseado em α, funciona para qualquer par (n, f).

Mais recentemente, [Khouri and Greve 2014b] apresentaram um protocolo para implementar Ω num modelo de memória compartilhada que, semelhantemente ao proposto aqui, tem por base um padrão de acesso à memória compartilhada. Até onde sabemos, não existe outra implementação de Ω para memória compartilhada baseada em suposições livre de tempo. Nesse sentido, os dois trabalhos são pioneiros.

Entretanto, [Khouri and Greve 2014b] apresentam um protocolo pouco eficiente, cujos processos (todos) continuam a escrever nos registradores para sempre, mesmo após a eleição do líder. O protocolo aqui apresentado avança na obtenção de uma solução ótima para as escritas, reunindo condições para que somente o líder, após um tempo, continue a escrever no seu registrador. Com tal estratégia, semelhantemente às soluções [Aguilera et al. 2004, Malkhi et al. 2005, Hutle et al. 2009] para passagem de mensagens, que visavam optimalidade na quantidade de canais síncronos e de processos emissores de mensagens, avançamos no estado da arte da implementação de Ω para memória compartilhada. Em ambos os casos, o objetivo de fazer com que somente o líder atue (emita mensagens ou escreva no registrador, após um tempo) é atingido.

7. Considerações Finais

Este artigo avança na compreensão do problema de eleição de líder em memória compartilhada, considerando um sistema dinâmico sem requisitos temporais. Em resumo, as suas principais contribuições são: (i) A definição de um modelo de sistema dinâmico sujeito a uma propriedade livre de tempo baseada num padrão de acesso à memória, que permite a implementação de um serviço de líder numa memória compartilhada assíncrona, onde o conjunto de participantes é desconhecido; (ii) Um protocolo que implementa um serviço de líder da classe Ω adequado ao modelo proposto e que visa optimalidade na escrita dos registradores. O sistema dinâmico considera a existência de um conjunto de processos estáveis (entram no sistema e permanecem ativos sem falhar ou sair), cuja cardinalidade mínima é α. O algoritmo proposto tem resiliência ótima, já que mantém a corretude se ao menos um único processo estável permanece no sistema ($\alpha \geq 1$). Como trabalho futuro pretende-se avaliar a possibilidade de uso de registradores regulares ao invés de atômicos, já que os primeiros são mais fracos.

Referências

Aguilera, M., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. (2004). Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 328–337, New York, NY, USA. ACM.

Aguilera, M. K. (2004). A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59.

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. (2001). Stable leader election. In Welch, J., editor, *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg.

Aguilera, M. K., Englert, B., and Gafni, E. (2003). On using network attached disks as shared memory. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 315–324, New York, NY, USA. ACM.

Arantes, L., Greve, F., Sens, P., and Simon, V. (2013). Eventual leader election in envolving mobile networks. In *Proceedings of the 17th International Conference on Principles of Distributed Systems*, OPODIS '13, Berlin, Heidelberg. Springer-Verlag.

Baldoni, R., Bonomi, S., Kermarrec, A.-M., and Raynal, M. (2009). Implementing a register in a dynamic distributed system. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 639–647.

Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722.

Fernández, A., Jiménez, E., and Raynal, M. (2006). Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In *In DSN*, pages 166–175.

Fernández, A., Jiménez, E., and Raynal, M. (2007). Electing an eventual leader in an asynchronous shared memory system. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 399–408.

Fernández, A., Jiménez, E., Raynal, M., and Trédan, G. (2010). A timing assumption and two t-resilient protocols for implementing an eventual leader service inÂ asynchronous shared memory systems. *Algorithmica*, 56(4):550–576.

Fischer, M. J., Lynch, N. A., and Paterson, M. D. (1985). Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382.

Gafni, E. and Lamport, L. (2003). Disk paxos. *Distrib. Comput.*, 16(1):1–20.

Guerraoui, R. and Raynal, M. (2006). A leader election protocol for eventually synchronous shared memory systems. In *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on*, pages 6 pp.–.

Guerraoui, R. and Raynal, M. (2007). The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67.

Hutle, M., Malkhi, D., Schmid, U., and Zhou, L. (2009). Chasing the weakest system model for implementing ω and consensus. *IEEE Transactions on Dependable and Secure Computing*, 6(4):269–281.

Jiménez, E., Arévalo, S., and Fernández, A. (2006). Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2):60 – 63.

Khouri, C. and Greve, F. (2014a). Eleicão assíncrona de líder em memória compartilhada dinâmica. In *Relatório Técnico*, number 12, Salvador, BA. DCC/UFBA.

Khouri, C. and Greve, F. (2014b). Servico de líder em sistemas dinâmicos com memória compartilhada. XIV WTF-SBRC, Porto Alegre, RS, Brasil. SBC.

Lamport, L. (1986). On interprocess communication. *Distributed Computing*, 1(2):77–101.

Lo, W.-K. and Hadzilacos, V. (1994). Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *Proceedings of the 8th International Workshop on Distributed Algorithms*, WDAG '94, pages 280–295, London, UK. Springer-Verlag.

Malkhi, D., Oprea, F., and Zhou, L. (2005). ω meets paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 199–213, Berlin, Heidelberg. Springer-Verlag.

Mostefaoui, A., Mourgaya, E., and Raynal, M. (2003). M.: Asynchronous implementation of failure detectors. In *In: Proc. International IEEE Conference on Dependable Systems and Networks (DSN'03*, pages 351–360.

Mostefaoui, A., Mourgaya, E., Raynal, M., and Travers, C. (2006). A time-free assumption to implement eventual leadership. *Parallel Processing Letters*, 16(02):189–207.