Serviço de Líder em Sistemas Dinâmicos com Memória Compartilhada

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

[1]Departamento de Ciências Exatas – Univ. Estadual do Sudoeste da Bahia
Vitória da Conquista, BA – Brasil

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

catia091@dcc.ufba.br, fabiola@dcc.ufba.br

Abstract. *In spite of the importance of dynamic distributed environments, such as SANs (storage area networks) and multicore architectures, we found very few proposals of models and protocols for implementing eventual leader election. This abstraction is fundamental to the implementation of consistency and fault-tolerance requirements. Most approaches for electing a leader consider static systems, where processes communicate by message passing, satisfying timing constraints. This paper presents a leader election service, of class* Ω*, for a* dynamic *asynchronous system subject to crash failures, where processes communicate through* atomic shared registers *following a memory access pattern, free of temporal requirements.*

Resumo. *Apesar da importância que ambientes dinâmicos como as SANs (redes de área de armazenamento) e as arquiteturas multi-núcleo ocupam no cenário atual dos sistemas distribuídos, poucas são as propostas de modelos e protocolos para a implementação de eleição de líder após um tempo nesses contextos. Essa abstração é fundamental para a implementação dos requisitos de consistência e tolerância a falhas desses sistemas. Entretanto, a maioria das abordagens de eleição de líder considera sistemas estáticos, onde os processos se comunicam por troca de mensagens, satisfazendo requisitos temporais. Este artigo apresenta um serviço de eleição de líder, da classe* Ω*, para um* sistema dinâmico *assíncrono, sujeito a falhas por parada, em que os processos se comunicam através de* registradores atômicos compartilhados *e segundo um padrão de acesso à memória, livre de requisitos temporais.*

1. Introdução

Garantir tolerância a falhas em sistemas distribuídos assíncronos não é uma tarefa trivial. Na realidade, é mesmo impossível distinguir se um processo falhou ou se apenas está muito lento quando não existem limites definidos para a velocidade de passos de processos ou para o tempo de transmissão de mensagens. Para contornar essa dificuldade, uma alternativa é estender o sistema com a utilização de detectores de falhas não confiáveis [Chandra and Toueg 1996]. Estes mecanismos funcionam como oráculos distribuídos que fornecem informações sobre processos falhos no sistema.

Um desses detectores largamente utilizado é o *detector de líder após um tempo* (do inglês: *eventual leader detector*), conhecido como Ômega (ou Ω) [Chandra et al. 1996].

Num sistema distribuído confiável, camadas superiores de serviço tais como consenso, difusão atômica, replicação de máquinas de estado, etc., contam com um serviço de líder após um tempo para prover os processos com uma primitiva que retorna a identidade de um líder, i.e., um processo correto no sistema. Embora possa fornecer informações equivocadas durante um intervalo de tempo finito, após o período de instabilidade, todo processo do sistema recebe a mesma identidade de líder. Diz-se então que todo processo no sistema confia naquele eleito como líder.

Um atributo importante de um detector Ω é o de permitir o projeto de algoritmos *indulgentes* [Guerraoui and Raynal 2003]. Nesse caso, os algoritmos podem conviver pacificamente com os erros do detector, garantindo que suas propriedades de segurança (ou *safety*) não sejam violadas durante o período instável. A terminação (ou *liveness*) de tal algoritmo será então viabilizada pelo detector quando todos os processos confiarem no mesmo líder.

A classe de detectores Ω é a mais fraca a possibilitar a resolução do consenso em sistemas assíncronos sujeitos a falhas [Chandra et al. 1996]. Entretanto, a implementação desse detector em si também não é possível em sistemas puramente assíncronos (pelas razões colocadas acima). É preciso enriquecer tal sistema com suposições a respeito do comportamento dos processos e/ou canais de comunicação. Nesse sentido, vários trabalhos têm sido publicados que propõem propriedades adicionais a sistemas assíncronos e os algoritmos correspondentes que implementam o serviço de líder nessas condições. Contudo, a maioria desses trabalhos é voltada para sistemas em que o conjunto de participantes é estático, conhecido, e a comunicação se dá através de troca de mensagens.

Este artigo trata da implementação de um serviço de líder em um *sistema dinâmico* assíncrono sujeito a falhas por parada (*crash*), em que os processos se comunicam através de uma *memória* constituída de *registradores atômicos compartilhados*. A nossa motivação vem da proliferação de sistemas como redes de dispositivos móveis *ad-hoc*, computação em nuvem, sistemas p2p, redes de área de armazenamento (SAN's), etc. Nesses sistemas, considerados dinâmicos, os nós podem entrar e sair arbitrariamente, de modo que o conjunto de participantes é auto-definido e variável com a execução. Normalmente, no início de uma execução, cada nó conhece apenas sua identidade, e sabe que as identidades dos demais nós são distintas. Esse dinamismo introduz uma nova fonte de imprevisibilidade no sistema e deve ser levado em conta no projeto dos algoritmos.

O modelo de interação por memória compartilhada adequa-se bem a sistemas em que discos conectados em rede (*network attached disks*) aceitam requisições de leitura e escrita propiciando a comunicação entre computadores. 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. Os trabalhos em [Gafni and Lamport 2003, 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 que tais ambientes ocupam no cenário atual dos sistemas distribuídos, poucas são as propostas de modelos e protocolos para a implementação de

Ω nessas circunstâncias. A maioria das abordagens considera sistemas estáticos de passagem de mensagens que fazem suposições adicionais relativas à sincronia de processos e/ou canais de comunicação. Em geral, essas propostas sugerem que todos ou parte dos canais funcionam "em tempo" (*timely*) a partir de um determinado instante e variam essencialmente no teor de sincronia que exigem para convergir. Outra abordagem, menos frequente, baseia-se num *padrão de mensagens* ao invés de assumir limites para processos ou transferência de mensagens. Ao contrário da primeira, protocolos baseados nesta última abordagem não utilizam temporizadores e são por isso chamados livres de tempo (ou *time-free*). Poucos são os protocolos para sistemas dinâmicos de memória compartilhada e, até onde sabemos, nenhum baseia-se em alguma abordagem livre de tempo.

Nesse artigo apresentamos um protocolo de detecção de líder após um tempo, da classe Ω, para um sistema dinâmico assíncrono, onde processos se comunicam através de memória compartilhada, seguindo um padrão de acesso à memória e portanto uma abordagem livre de tempo. Nesse sentido, nosso trabalho é pioneiro.

2. Modelo de Sistema para Eleição de Líder

O sistema dinâmico é composto por um conjunto infinito Π de processos que se comunicam através de uma memória compartilhada e que podem falhar (*crashing*), parando prematura ou deliberadamente (por exemplo, saindo arbitrariamente do sistema). Não fazemos suposições sobre a velocidade relativa dos processos, para dar passos ou para realizar operações de acesso à memória compartilhada, isto é, o sistema é assíncrono. Para facilitar a apresentação, assumimos a existência de um relógio global que não é acessível pelos processos. A faixa de ticks do relógio, $\mathcal{T}$, é o conjunto dos números naturais.

2.1. Processos

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 conjunto de participantes $\Pi = \{p_1, ..., p_n\}$ é limitado. Π (inclusive n) pode variar de execução a execução, mas existe um instante a partir do qual nenhum novo processo entra (por isso o nome "*chegada finita*"). Cada processo localiza-se em um nó distinto e cada um deles possui uma identidade distinta tal que o conjunto dessas identidades é totalmente ordenado (denotamos i a identidade do processo p_i). Um processo p_i que deseja entrar no sistema não têm conhecimento sobre o conjunto de participantes e o estado dos registradores compartilhados e nem mesmo possui uma identidade.

Em um sistema de passagem de mensagens em que o conjunto de participantes é desconhecido, os processos podem, por exemplo, começar a executar o algoritmo de Ω e à medida que o algoritmo progride, os nós aprendem dinamicamente sobre a existência dos outros participantes, através de trocas de mensagens, construindo assim o *membership*. No nosso modelo de memória compartilhada, abstraímos essa construção do conjunto de participantes delegando essa tarefa à implementação da memória compartilhada subjacente. Assim, para entrar no sistema, p_i invoca uma operação *join()* a fim de inteirar-se do estado do sistema, criar seus registradores e obter uma identidade (distinta da dos demais participantes). Além disso, como será visto mais adiante, a operação $join()$ se encarrega de garantir que os processos que já se encontram no sistema criem registradores que armazenarão informações sobre p_i com um estado inicial que favoreça a manutenção de um

líder correto (estável) eleito. Um processo então só pode começar a participar ativamente da computação e concorrer à posição de líder após o término da operação $join()$.

Um processo que segue a especificação do algoritmo e não falha em uma execução é dito *correto*. O padrão de falhas do sistema, $F(t)$, é o conjunto de processos que entraram no sistema, mas falharam até o instante t, e um processo pertencente a $F(t)$, para algum t, é dito *faltoso*. Considerando a natureza dinâmica do sistema, denotamos $ATIVO(t)$ o conjunto de processos que entraram no sistema e não saíram (nem falharam) até o instante t. Se um processo sai (ou falha) e volta para a computação, ele é considerado um novo processo e portanto terá uma nova identidade.

A despeito da movimentação de entrada e saída de processos, para que se possa realizar alguma computação útil, é preciso que exista um período de estabilização do sistema bem como um conjunto de processos que permaneça ativo neste período. Definimos então o conjunto de processos *estáveis* a partir da definição do conjunto de processos ativos. Informalmente, um processo correto que nunca deixa a computação, ou seja, um processo que se ligou ao sistema num instante t e permanece ativo $\forall t' \geq t$ é dito *estável*.

Definição 1 (STABLE) *STABLE* = $\{p_e : \exists t \in \mathcal{T}, \forall t' \geq t : p_e \in ATIVO(t')\}$

Para garantir progresso, o tamanho do conjunto de processos estáveis deve ser limitado inferiormente. No contexto deste trabalho, uma vez que não se conhece *a priori* nem n (cardinalidade do sistema) nem f (máximo número de processos falhos), utilizaremos o parâmetro α para denotar o número mínimo de processos que devem estar vivos no período estável. Assim, uma condição necessária para a terminação (*liveness*) neste modelo é: $|STABLE| \geq \alpha$, onde α abstrai qualquer par (n, f). No modelo clássico de sistemas distribuídos, α corresponde ao valor $(n - f)$.

2.2. Memória Compartilhada

Um sistema de 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. Um objeto compartilhado possui um *tipo*, que define o domínio de valores que podem ser armazenados; e um conjunto de operações que são a única forma de acessar o objeto. No modelo em questão esses objetos são *registradores atômicos* multivalorados do tipo 1-*escritor/n-leitores* (1W/nR), que se comportam corretamente. Um registrador multivalorado 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 compartilhado modela uma forma de comunicação persistente onde o emissor é o escritor, o receptor é o leitor, e o estado do meio de comunicação é o valor do registrador. Comportamento *correto* significa que o registrador sempre pode executar uma leitura ou escrita e nunca corrompe seu valor. 1W/nR significa que o registrador pode ser escrito por um processo (seu proprietário) e lido por múltiplos processos.

Em um sistema com registradores atômicos, embora as operações sobre um mesmo registrador possam se sobrepor em um intervalo de tempo, elas parecem ser instantâneas, de modo que para qualquer execução existe alguma forma de ordenar totalmente leituras e escritas como se elas ocorressem instantaneamente em algum ponto entre sua invocação e resposta. Assim, o valor retornado por uma leitura que sobrepõe uma

ou mais escritas é o mesmo que seria retornado se não houvesse sobreposição. Dizemos então que um registrador atômico é *linearizável* [Herlihy and Wing 1990].

3. Especificação de Ω

Já é bem estabelecido que não existe solução determinística para alguns problemas fundamentais em sistemas distribuídos assíncronos sujeitos a falhas [Fischer et al. 1985]. Informalmente, essa impossibilidade é explicada pela dificuldade de se distinguir, em determinado instante, se um processo parou ou está excessivamente lento. Para contornar essa dificuldade, uma alternativa é estender o sistema com mecanismos detectores (de falhas ou de líder) tais como os propostos por [Chandra and Toueg 1996]. Tais dispositivos são constituídos de módulos distribuídos que têm o objetivo de prover o sistema com "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 invocaram a função. Num contexto de sistema dinâmico, um detector Ω satisfaz assim 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 o mesmo processo $p_l \in STABLE$.

4. Eleição de Líder em um Sistema Dinâmico de Memória Compartilhada

Um sistema dinâmico, no qual processos podem entrar e sair arbitrariamente, apresenta uma dificuldade adicional no projeto de algoritmos tolerantes a falhas. Por exemplo, numa implementação de Ω pode acontecer de nenhum processo permanecer por tempo suficiente para ser eleito. Desse modo, é preciso que o sistema satisfaça alguma propriedade de estabilidade para que a computação progrida e então termine.

A abordagem utilizada neste trabalho é baseada num *padrão de acesso à memória* compartilhada, de certo modo semelhante à introduzida em [Mostefaoui et al. 2003] para sistemas de passagem de mensagem. Informalmente, assumimos que existe um conjunto de α processos que permanecem vivos por um período de tempo suficiente para eleger o líder. Esse valor de α corresponde ao valor $(n - f)$, em sistemas estáticos, que representa um limite inferior para o número de processos que precisam interagir para garantir progresso e terminação a certos protocolos.

4.1. Propriedade do Sistema

No algoritmo a ser apresentado, com o intuito de informar aos demais processos do sistema que está ativo, o processo p_i se mantém incrementando um contador num registrador compartilhado, de nome $Alive[i]$. Por outro lado, periodicamente, os demais processos p_j consultam o valor em $Alive[i]$ para saber se p_i progrediu. Assim, $\forall p_j, \forall p_i$: p_j lê $Alive[i]$ e, caso o valor lido seja igual àquele lido anteriormente, então p_j pune p_i, suspeitando que ele não esteja mais ativo no sistema. Chamaremos $vencedores_j(t)$ ao conjunto dos primeiros α processos cujos registradores p_j percebeu estarem atualizados com relação à sua última leitura. Formalizamos então a propriedade que habilita o progresso e término do algoritmo de líder a ser apresentado.

Propriedade 1 (Padrão de Acesso à Memória – PAM) *Existe um instante t e um processo $p_l \in STABLE$ tais que, $\forall t' \geq t$, $\forall p_j \in STABLE : p_l \in vencedores_j(t')$.*

Algorithm 1 EVENTUAL LEADER ELECTION (*code for p_i*)

```
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: ALIVE
(4)  repeat forever
(5)      alive_i ← alive_i + 1;
(6)      write (Alive[i], alive_i);

TASK 2: PUNISHMENT
(7)  repeat forever
(8)      updated_i ← {i};
(9)      while (|updated_i| < α) do
(10)         for each (j) do:
(11)             if (j ∉ updated_i) then {
(12)                 read (Alive[j], alive_i^j);
(13)                 if (alive_i^j ≠ last_alive_i[j]) then {
(14)                     last_alive_i[j] ← alive_i^j;
(15)                     updated_i ← updated_i ∪ j; } }
(16)     for each (j ∉ updated_i) do:
(17)         read (Punishments[i][j], pun2_i);
(18)         write(Punishments[i][j], pun2_i + 1));

leader()
(19) for each j do:
(20)     soma_i[j] ← 0;
(21)     for each k do:
(22)         read(Punishments[k][j], pun1_i);
(23)         soma_i[j] ← soma_i[j] + pun1_i;
(24) leader_i ← Min(soma_i[j], j);
(25) return leader_i;
```

4.2. Visão Geral do Algoritmo de Eleição de Líder

O Algoritmo 1 é executado em paralelo por todos os processos. Um processo p_i define como seu líder o processo que ele considera "menos suspeito". Essa ideia de suspeição é expressa através de contadores indexados. Sempre que p_i suspeita de p_j, incrementa o contador correspondente. Assim, define-se um arranjo de registradores tal que $Punishments[i][j]$ representa o número de vezes que p_i suspeitou (e puniu) p_j.

O nível de suspeição de um processo p_i com relação a um processo p_j será definido pela soma dos contadores: $soma_i[j] = \sum_{\forall k} Punishments[k][j]$. O processo escolhido como líder será o p_j para o qual $soma_i[j]$ seja mínima. Se houver mais de uma soma com valor igual ao mínimo, o desempate será definido pela menor identidade de processo, isto é, será considerada a ordem lexicográfica. Dados os pares $(soma_i[x], p_x)$ e $(soma_i[y], p_y)$ com $x < y$, $Min((soma_i[x], p_x), (soma_i[y], p_y)) = p_x$, se $soma_i[x] \leq soma_i[y]$; e p_y , se $soma_i[y] < soma_i[x]$.

Quando um processo p_r quer entrar no sistema, invoca a operação $join()$ que provê um estado consistente dos registradores para p_r; inicializa os registradores $Alive[r]$ e $Punishments[r][i]$, $\forall i$; e aloca espaço para seus arranjos dinâmicos. Os demais processos p_i, ao receberem um solicitação de $join()$ da parte de p_r, criam seus registradores $Punishments[i][r]$ devidamente inicializados; e alocam espaço em seus arranjos dinâmicos locais para acomodar p_r. Apenas após o retorno de $join()$ um processo p_r estará apto a participar da computação e será considerado pelos demais processos no sistema como um candidato a líder (incluindo os registradores correspondentes na escolha).

Para impedir que um líder estável seja demovido por um recém-chegado, cada registrador $Punishments[i][r]$ é inicializado com o valor do contador de punições de p_i atribuído a p_l acrescido de 1: $Punishments[i][r] \leftarrow Punishments[i][l] + 1$. Dessa forma, o nível de suspeição inicial do recém-chegado ($\sum_{\forall i} Punishments[i][r]$) será maior que o do líder corrente.

No algoritmo são utilizados os seguintes registradores compartilhados:

- $Alive[i]$ – contador responsável por indicar o progresso de p_i, inicializado com 0.
- $Punishments[i][j]$ – contador pertencente a p_i que registra o número de punições aplicadas por p_i a p_j. Inicializado com $Punishments[i][l]+1$ (o contador referente ao líder corrente acrescido de 1).

Um processo p_i também utiliza as seguintes variáveis locais:

- $alive_i$ – contador que controla o progresso de p_i cujo valor é escrito em $Alive[i]$. Inicializado com 0.
- $alive_i^j$ – variável auxiliar usada para leitura de $Alive[j]$.
- $last\_alive_i[1..]$ – vetor dinâmico que armazena o último valor lido em $Alive[1..]$. Cada entrada é inicializada com 0.
- $soma_i[1..]$ – vetor dinâmico que armazena a soma dos contadores de punições de cada p_j.
- $updated_i$ – conjunto utilizado para controlar quais processos tiveram seu progresso verificado.
- $pun1_i$, $pun2_i$ – variáveis auxiliares usadas para leitura de $Punishments[i][j]$.

4.3. Descrição do Algoritmo

Para ligar-se ao sistema, p_i invoca a operação $join()$ (linha 1), e ao retornar inicia três tarefas em paralelo (linha 2).

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

TASK 1: ALIVE – Ao executar essa tarefa, p_i informa intermitentemente aos demais participantes que ele está vivo com o fim de evitar falsas suspeitas a seu respeito. Ele faz isso incrementando seu registrador compartilhado $Alive[i]$ (linhas 4-6).

TASK 2: PUNISHMENT – Enquanto executa esta tarefa, p_i se mantém verificando se os registradores $Alive[j]$ foram atualizados desde sua última leitura (linhas 7-15). Para fazer isso, a cada iteração do laço mais externo (linha 7), p_i guarda o valor lido em cada $Alive[j]$ no vetor local $last\_alive_i[j]$, a fim de poder compará-lo com a leitura na próxima iteração. Como p_i nunca pune a si mesmo, inicialmente ele atribui sua própria identidade à variável conjunto local $updated_i$ que ele utiliza para controlar quais registradores são achados atualizados (linha 8). Então ele procura por pelo menos α processos p_j que tenham atualizado seus registradores (linhas 9-10). Para cada um desses (linhas 10-13), p_i guarda o valor atualizado de $Alive[j]$ e inclui p_j em $updated_i$ (linhas 14-15).

A partir daí, p_i pune cada processo p_j que não foi encontrado "atualizado". Para isso ele adiciona 1 ao valor corrente no registrador $Punishments[i][j]$ (linhas 16-18). Assim, os primeiros α processos que atualizarem seus registradores $Alive[j]$ não são punidos.

leader() – Esta função retorna uma identidade de processo que, do ponto de vista de p_i é correto e considerado líder por ele. Para a escolha do líder, a soma das punições atribuídas a cada p_j é acumulada em $soma_i[j]$ (linhas 19-23). É escolhido como líder o processo com menor soma, utilizando a identidade como desempate (linhas 24-25).

4.4. Prova de Correção

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

Prova. As punições ocorrem em TASK 2: PUNISHMENT. O comportamento de p_j vai depender do fato dele pertencer ou não ao conjunto $STABLE$. (1) Se $p_j \in STABLE$, de acordo com a Propriedade 1, existem um processo p_l e um instante t', tais que $\forall t'' \geq t'$, p_j encontrará $Alive[l]$ atualizado com relação à sua última leitura e portanto avaliará o predicado da linha 13 como verdadeiro e incluirá p_l em $updated_i$ (linha 15). Por conseguinte, p_l não será punido a partir de t' (linhas 16-18). (2) Se $p_j \notin STABLE$ é porque ele falha ou deixa o sistema em algum instante t'''. Em ambos os casos, p_j deixa de dar passos e portanto não pune p_l a partir de $t \geq t'$, $t \geq t'''$. $_{Lema1}\square$

Lema 2 *Todo processo que falha ou deixa o sistema em algum instante t, será permanentemente punido por qualquer processo estável a partir de algum $t' \geq t$.*

Prova. Uma vez que um processo p_j falha ou abandona o sistema em um instante t, ele para de executar TASK 1: ALIVE (linhas 4-6). Portanto, a partir de um instante $t' \geq t$, toda vez que um processo estável p_e executar TASK 2, encontrará $Alive[j] = last\_alive_e[j]$ (linhas 12-13) e acabará punindo p_j. Uma vez que $p_e \in STABLE$, permanecerá executando a tarefa e punindo p_j. $_{Lema2}\square$

Lema 3 *Um processo p_j que falha ou deixa o sistema em um instante t jamais será eleito líder a partir de algum instante $t' \geq t$.*

Prova. (1) Pelo Lema 2, p_j será permanentemente punido por todo processo estável p_e a partir de algum $t'' \geq t$. Isso fará com que, a partir deste instante, os contadores $Punishments[e][j]$, $\forall p_e \in STABLE$, permaneçam crescendo monotonicamente (linhas 7-18). (2) Pelo Lema 1, existe um instante t''' após o qual algum $p_l \in STABLE$ jamais é punido. Portanto, a partir de t''' os contadores $Punishments[-][l]$ permanecem constantes. De (1) e (2) temos que a partir de um instante t', $t' \geq t''$ e $t' \geq t'''$,

$\sum_{\forall k} Punishments[k][j] > \sum_{\forall k} Punishments[k][l]$, o que impedirá que p_j seja escolhido líder (linhas 24-25). $Lema3\square$

Lema 4 *Se um processo $p_l \in STABLE$ tiver sido eleito líder em um instante t, um processo p_r que chega ao sistema em um instante $t' > t$ não demove o líder eleito.*

Prova. Quando um processo p_r entra no sistema, cada registrador $Punishments[k][r]$ é inicializado com o valor $Punishments[k][l] + 1$ (linha 3). O líder eleito p_l, por sua vez, é tal que $l = Min(\sum_{\forall k} Punishments[k][j], j)$ (linha 24). Portanto, uma vez que a soma dos contadores relativos a p_r, $\sum_{\forall k} Punishments[k][r]$, será maior que a soma dos contadores relativos a p_l, p_r jamais será atribuído a $leader_i$. $Lema4\square$

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

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

5. Trabalhos Relacionados

A maioria das propostas de implementação de Ω em sistemas assíncronos apresentadas na literatura considera sistemas de passagem de mensagens. Nesse contexto, a busca por requisitos mínimos que possibilitem a implementação segue duas vertentes distintas: abordagens baseadas em tempo (*timer-based*) e abordagens baseadas em padrão de mensagem (*message-pattern*). A Tabela 1 resume as principais abordagens e suas características.

Abordagem	Comunicação	Π	Conectividade	Propriedade do Modelo
[Chandra et al. 1996]	pass. de mensagens	conhecido	grafo completo	todos os canais *eventually timely*
$\exists p$ *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 $\Diamond f$*-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 $\Diamond f$*-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 alcançado de p por canais *eventually timely*
[Fernández et al. 2006]	pass. de mensagens	desconhecido dinâmico	nós corretos conectados	$\exists p$ correto e $> (n\text{-}f)\text{-}\alpha + 1$ processos corretos alcançáveis de p por canais *eventually timely*
[Guerraoui and Raynal 2006]	memória compartilhada	conhecido	-	sistema síncrono após um tempo
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.

Table 1. Abordagens para Implementação de Ω

5.1. Abordagens Baseadas em Tempo

Estão relacionadas a suposições adicionais sobre limites para os passos de processos ou para transferência de mensagens. Geralmente, esses limites não precisam ser conhecidos

a priori além de que não precisam ser verificados durante todo tempo de execução do sistema. Os protocolos para sistemas assíncronos baseados nessa abordagem utilizam *time-outs* e diferem entre si, basicamente, na quantidade de conectividade e sincronia exigida para conversão. As nove primeiras entradas da Tabela 1 correspondem a essa 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]. A proposta de [Aguilera et al. 2001], considera que existem 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*). Tal processo p é dito ser *eventually accessible*.

Em [Aguilera et al. 2004], a definição de *eventually timely link* é por eles apresentada juntamente com a de outras propriedades referentes a canais. *Integridade*: um processo p recebe uma mensagem m de q apenas uma vez e somente se p enviou m para q. Um canal é *confiável* (*reliable*) se ele satisfaz integridade além de que, se p envia uma mensagem m a um processo correto q, então q acaba por receber m. Se um canal satisfaz *justiça* (*fairness*), então, se um processo p envia um número infinito de mensagens de um determinado tipo, então o canal entrega um número infinito de mensagens daquele tipo. Um canal é *fair-lossy* se ele satisfaz integridade e *fairness*: se um processo p envia um número infinito de mensagens de um determinado tipo para q, e q é correto, então q recebe de p um número infinito de mensagens daquele tipo. Um canal é *timely* se satisfaz integridade e existe um δ tal que se p envia uma mensagem m a q no instante t, e q é correto, então q recebe m de p até o instante $t + \delta$.

Finalmente, um canal é *eventually timely* se satisfaz integridade e existem t e δ tais que se p envia uma mensagem m a q no instante $t' \geq t$, e q é correto, então q recebe m de p até o instante $t' + \delta$ (mensagens enviadas antes de t podem ser perdidas). Os autores então mostram que Ω pode ser implementado em sistemas com n processos dos quais até f podem falhar (*crash*), cujos canais sejam *fair-lossy*, desde que pelo menos um processo correto p possua f canais de saída que sejam *eventually timely*. Nesse caso, p é dito ser um $\Diamond f\text{-}source$. Adicionalmente, os autores mostram que se os canais forem confiáveis (*reliable*), é possível implementar um protocolo otimal, do ponto de vista da comunicação. Isso porque o limite inferior para o número de canais que precisam carregar mensagens para sempre em qualquer implementação de Ω livre de falhas é f.

Em [Malkhi et al. 2005], a estratégia adotada é diferente das anteriores; eles usam a noção de *eventually f-accessibility*. Em um sistema com n processos em que até $f \leq (n-1)/2$ podem falhar por parada e cujos canais são confiáveis, existe um limite superior δ para um *round-trip* de mensagens que é conhecido mas que não se mantém para todo canal em todo instante. Supõe-se que existem um instante t, um processo p e um conjunto Q de f processos, tais que, $\forall t' \geq t$, p troca mensagens com cada q dentro do limite δ, isto é, p é $\Diamond f$*-accessible*. No artigo é apresentado um protocolo baseado na suposição. Deve-se observar que o subconjunto de processos Q 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. A diferença básica da suposição anterior é que aqui só é exigido que os canais de saída de p sejam *eventually timely*. É assumido que um processo faltoso sempre recebe as mensagens em tempo. O trabalho mostra que a complexidade de comunicação para implementações baseadas nessa premissa é limitada inferiormente por (nt) e propõe três protocolos, dos quais um apresenta essa complexidade.

Como citado na Introdução, o projeto de um algoritmo para um sistema dinâmico precisa levar em conta a imprevisibilidade devida ao fato de que processos podem entrar e sair arbitrariamente do sistema. Obviamente, uma característica inerente a tais ambientes é a falta de conhecimento inicial que cada nó tem a respeito do conjunto de participantes. Uma abordagem tradicional em tais sistemas, quando a comunicação se dá por passagem de mensagem, é a utilização de primitivas de difusão a partir de cujas respostas os processos constroem seu conhecimento (*membership*) do sistema, ao passo em que também constroem o serviço de líder.

[Jiménez et al. 2006] propõem um protocolo para ambiente dinâmico com troca de mensagens, supondo que existe um limite inferior sobre o número de passos por unidade de tempo que um processo dá e que todos os processos corretos são conectados por canais *eventually timely*. Diferentemente do nosso, o protocolo apresentado é baseado em $timeouts$. [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. Essa constante α abstrai todos os pares (n, f) tradicionalmente consideradas como conhecidas, tal que $n - t \geq \alpha$. Dessa forma, um protocolo baseado em α, funciona para qualquer par (n, f). Diferentemente da nossa proposta, entretanto, os processos se comunicam por troca de mensagens e a conversão do algoritmo, baseado em *timeouts*, exige a existência de alguns canais *eventually timely*.

No que diz respeito ao paradigma de comunicação por memória compartilhada, muito poucas são as propostas encontradas na literatura para a implementação de Ω, a maioria delas para sistemas estáticos. [Guerraoui and Raynal 2006] implementam um protocolo com registradores regulares compartilhados, mas diferentemente do nosso trabalho, consideram um sistema estático, síncrono após um tempo. [Fernández et al. 2007, Fernández et al. 2010], assim como nós, consideram um sistema assíncrono com registradores atômicos compartilhados. Ao contrário da nossa proposta, entretanto, o ambiente é estático e eles fazem suposições adicionais sobre o sincronismo do sistema para atingir a conversão do protocolo, o qual é baseado em *timeouts*.

Em um sistema de memória compartilhada cujos participantes são desconhecidos, um processo que entra na computação, naturalmente, precisa inteirar-se do estado do sistema antes de começar a 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. No início da operação, p_i envia um *broadcast* com sua identidade informando aos demais processos sobre sua chegada. Os processos p_j respondem, informando sua identidade e o valor local de seu registrador. p_i então processa os dados recebidos, definindo um valor consistente para o registrador.

Ao terminar $join()$, p_i está apto a realizar operações $read$ e $write$ sobre o objeto compartilhado.

De modo semelhante, o nosso protocolo envolve a invocação de uma operação $join()$ que se encarrega de criar registradores devidamente inicializados, fornecer ao recém-chegado, p_r, uma cópia consistente de cada registrador compartilhado, e incluí-lo na computação. A implementação da operação $join()$, entretanto, está fora do escopo do nosso trabalho. O algoritmo considera que ao concluir a operação, p_r estará apto a trocar informações com os demais participantes através da memória compartilhada e vice-e-versa. Além disso, o estado inicial de p_r será tal que impeça a demoção equivocada de um líder estável.

Outras implementações de Ω disponíveis na literatura, assim como a nossa, não consideram suposições adicionais relativas à sincronia de processos ou canais de comunicação. Ao invés disso, baseiam-se em um padrão de troca de mensagens. Até onde sabemos, não existe uma implementação de Ω para sistemas assíncronos de memória compartilhada estendidos com suposições adicionais baseadas em tempo.

5.2. Abordagens Baseadas em Padrão de Mensagens

A fim de possibilitar a implementação de detectores de falhas em um sistema assíncrono, [Mostefaoui et al. 2003] introduziram uma nova abordagem. Eles se baseiam em um mecanismo *query-response*, assumindo que a resposta de algum processo p_i às últimas *queries* requisitadas por determinados processos p_j está sempre entre as primeiras $n - f$ respostas recebidas por cada p_j, onde n é o número total de processos no sistema e f um limite superior para o número de falhas. Eles mostram que se essa suposição é satisfeita a partir de um instante $t \geq 0$, um detector $\Diamond S$ pode ser implementado. 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 em uma *sincronia lógica* capaz de contornar a impossibilidade de conversão do algoritmo [Mostefaoui et al. 2006].

Considerando uma suposição semelhante, [Arantes et al. 2013] propõem um modelo para implementação de Ω em um sistema dinâmico de passagem de mensagens que leva em conta a mobilidade dos nós durante a computação. Para garantir a corretude do protocolo, eles consideram uma versão da suposição baseada em padrão de mensagens adaptada a sistemas dinâmicos segundo a qual os processos envolvidos devem pertencer a um conjunto de nós *estáveis*, isto é, processos que, uma vez que se ligaram ao sistema no instante t, permanecem para sempre. Além disso, eles utilizam o limite inferior α sobre o número de processos corretos para controlar o número de respostas consideradas entre as primeiras. No protocolo aqui apresentado, introduzimos uma suposição semelhante, isto é, o modelo considera uma noção de sincronia lógica, porém com relação aos acessos à memória compartilhada.

6. Considerações Finais

Os resultados aqui reportados trazem uma primeira contribuição ao estudo do problema de líder em ambientes dinâmicos de memória compartilhada. Introduzimos uma propriedade comportamental que reflete um padrão relativo aos acessos feitos à memória compartilhada. No modelo considerado, a memória é constituída de registradores atômicos com-

partilhados *1-escritor-N-leitores*, tais que o registrador $R[i]$ é escrito pelo processo proprietário p_i e lido por qualquer processo p_j participante no sistema. Dessa maneira, cada processo p_i pode manifestar seu progresso incrementando $R[i]$ periodicamente. Cada p_j pode então conferir se cada p_i permanece vivo, observando o crescimento de $R[i]$.

A propriedade que propomos considera esse acompanhamento que cada p_j faz de cada $R[i]$. Informalmente, supomos que existe um processo *estável* p_l e um instante t, após o qual, as escritas de p_l em $R[l]$ estão sempre entre as primeiras α atualizações percebidas entre todos os registradores compartilhados, do ponto de vista de qualquer processo *estável* no sistema. A noção de *estável* refere-se a um processo que entra no sistema e permanece vivo (não falha ou deixa o sistema) durante toda computação. A existência de um conjunto de processos estáveis em um sistema dinâmico é necessária para que se possa realizar alguma computação útil. O parâmetro α representa o limite inferior para a cardinalidade deste conjunto.

Em resumo, as principais contribuições deste artigo foram:

- A definição de um modelo de sistema dinâmico assíncrono de memória compartilhada sujeito a uma propriedade baseada em um padrão de acesso à memória (*time-free*) que permite a implementação de um serviço de líder;
- Um protocolo que implementa um serviço de líder da classe Ω adequado ao modelo proposto.

Como trabalho futuro pretende-se avaliar a possibilidade de uso de registradores regulares ao invés de atômicos, já que os primeiros são mais fracos. Outra possibilidade é diminuir a complexidade do protocolo, de tal maneira que somente o líder tenha que escrever para sempre em seus registradores, enquanto os demais processos precisarão apenas ler.

References

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 evolving 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. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

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. (2003). The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53:2004.

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.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

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.

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.