Universidade Federal Fluminense

HIGOR DE PÁDUA VIEIRA NETO

Algoritmos Distribuídos para o Problema de Alocação de Múltiplos Recursos em Grids Computacionais

NITERÓI-RJ

2008

Universidade Federal Fluminense

HIGOR DE PÁDUA VIEIRA NETO

Algoritmos Distribuídos para o Problema de Alocação de Múltiplos Recursos em Grids Computacionais

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Processamento Paralelo e Distribuído.

Orientador:
Prof. Lúcia Maria A. Drummond, D.Sc.

NITERÓI-RJ

2008

Algoritmos Distribuídos para o Problema de Alocação de Múltiplos Recursos em *Grids* Computacionais

Higor de Pádua Vieira Neto

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Processamento Paralelo e Distribuído.

Aprovada por:

Profa. Lúcia Maria A. Drummond, DSc / IC-UFF
(Orientadora)

Prof. Eugene Francis Vinod Rebello, PhD / IC-UFF

Profa. Maria Clicia Stelling de Castro, DSc / IME-UERJ

Niterói, 18 de agosto de 2008.

Aos meus pais, por todo o amor, carinho e apoio.

Agradecimentos

Primeiramente agradeço à Deus, por ter me fortificado na fé nos diversos momentos de dificuldade e por ter me dado a luz nos caminhos mais escuros.

À minha família que sempre me deu suporte para que eu vencesse todos os meus desafios, aos quais também compartilho mais essa vitória. Meu pai, minha mãe, minha irmã e sobrinhas.

À minha namorada Geisiane, com quem sempre pude contar, sempre esteve ao meu lado me apoiando e me dando forças para que eu conseguisse alcançar meu objetivo.

À minha orientadora, Lúcia Drummond, pela valiosa orientação, fundamental para que eu fizesse esse trabalho.

À todos amigos conquistados no IC/UFF, em especial aos amigos Warley (toca) e Vinicius com quem compartilhei noites de estudos no laboratório, e também pela grande parceria e ajuda no desenvolvimento desse trabalho. Ao amigo Jacques pelo empenho em ajudar provendo alguns dos recursos necessários para os testes no laboratório.

Aos colegas de trabalho, em nome de Rodrigo C. Fernandes (Petrobrás), meus sinceros agradecimentos por toda ajuda e apoio recebidos, imprescindíveis para o desfecho desse trabalho.

Aos velhos amigos da Doctumtec e FIC de Caratinga, que me inspiraram e estiveram comigo em grande parte de minha história acadêmica e profissional. Também ao amigo e ex-professor/chefe Prof. Dr. Ulisses Leitão que acreditou em mim na hora que precisei dar esse grande passo em minha carreira.

À todas as pessoas que contribuíram direta ou indiretamente, e que fizeram parte dessa minha trajetória, Obrigado!

Resumo

Tipicamente, uma *Grid* é composta por uma coleção de *clusters*, cujo nós são conectados por enlaces dedicados de alta velocidade. A comunicação entre nós de *clusters* distintos é feita por *WANs* de baixa velocidade. Assim, é desejável que os algoritmos de sincronização sejam escaláveis e considerem tal topologia hierárquica de rede.

Neste trabalho é proposto um algoritmo baseado em *token* para resolver o problema da alocação de múltiplos recursos considerando a topologia hierárquica usual em ambientes de *Grid*.

Poucos trabalhos apresentam um algoritmo para exclusão mútua especificamente para *Grids*. Normalmente, eles consideram o compartilhamento de apenas um único recurso. Porém, é típico em *Grids* existir vários tipos de recursos compartilhados, incluindo hardware tais como RAM, espaço em disco, canais de comunicação e software tais como programas, arquivos e dados.

O algoritmo proposto foi comparado com um outro algoritmo baseado em *token* que resolve o problema de alocação de múltiplos recursos, mas que não considera as particularidades dos ambientes de *Grid*.

A redução no número de mensagens *inter-clusters* e no tempo de espera para entrar na seção crítica torna o algoritmo proposto bastante atrativo para aplicações de *Grid*.

Palavras-chave: Algoritmos Distribuídos, Exclusão Mútua, Grades Computacionais, Alocação Dinâmica de Recursos.

Abstract

Typically, a Grid is composed of a collection of clusters, with the nodes within each cluster being connected by high-speed dedicated links. Communication among nodes in distinct clusters is performed by much slower WANs. So, it would like to have scalable synchronization algorithms take into account such hierarchical network topology.

In this work, we propose a token-based algorithm to solve the multiple resource allocation problem considering the usual hierarchical network topology of Grid environments. Few papers present a mutual exclusion algorithm specifically for Grids and usually they consider the sharing of a unique resource. However, it is typical in Grids to have several kinds of shared resources, including hardware such as RAM, disk space, communication channels, and software such as programs, files and data.

We compared the proposed algorithm with a well known token-based algorithm for solving the multiple resource allocation problem, which does not consider the particularities of Grid environments.

The reduction in the number of inter-cluster messages and the waiting time to enter a critical section makes the proposed algorithm very attractive for Grid applications.

Keywords: Distributed Algorithms, Mutual Exclusion, Computational Grids, Dynamic Resource Allocation Problem.

Siglas e Abreviações

CT : *Control Token*;

SC : Seção Crítica;

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Muitas aplicações distribuídas requerem que um recurso seja alocado a somente um processo por vez. Cada processo possui um segmento de código, denominado de seção crítica (SC), na qual ele acessa os recursos compartilhados. O problema de coordenar a execução da seção crítica é resolvido usando o princípio da exclusão mútua, que provê acesso exclusivo aos recursos compartilhados em cada tempo.

Algoritmos que resolvem o problema da exclusão mútua em sistemas distribuídos podem ser basicamente divididos em dois grupos: algoritmos baseados em permissão (tais como [48], [46], [29] e [43]) e algoritmos baseados em *token* (tais como [17], [38], [36], [8] e [9]). No primeiro grupo, um processo requer a permissão de todos os outros processos ou da maioria deles para entrar em sua seção crítica. No segundo grupo, um único *token* é compartilhado pelos processos e a posse dele garante o direito de entrar em sua seção crítica. Algoritmos baseados em *token*, em sua maioria, consideram uma estrutura lógica dinâmica na forma de árvore, tornando-os adequados para ambientes hierárquicos, tais como *Grids*. Além disso, a maioria desses algoritmos apresenta uma média mais baixa de tráfego de mensagens considerando o número de processos quando comparado ao modelo baseado em permissão, que apresenta problemas de escalabilidade.

Grids são ambientes computacionais que possuem uma grande quantidade de nós espalhados sobre uma ampla área geográfica. Normalmente, uma *Grid* é composta por uma coleção de *clusters*, onde os nós são conectados através de enlaces dedicados de alta velocidade. A comunicação entre nós de *clusters* distintos é feita por canais WANs, que geralmente são muito mais lentos. Assim, para uma sincronização de recursos em larga escala, é necessário levar em conta tal topologia hierárquica [8]. Nesse trabalho, consideramos a topologia hierárquica como uma forma de organização e divisão dos processos em

domínios diferentes, com diferentes latências nos canais de comunicação. A Figura 1.1 dá uma visão abstrata de um ambiente *Grid* tal como foi tratada nesse trabalho.

Nesta dissertação de mestrado é proposto um algoritmo distribuído baseado em *token* para resolver o problema da alocação de M recursos, considerando uma topologia típica existente em ambientes de *Grid*. Considere que $R = (r_1, k_1), (r_2, k_2), \ldots, (r_m, k_m)$ seja o conjunto de recursos, tal que para cada recurso r_i existam k_i instâncias dele, que são compartilhadas pelo conjunto de processos $P = P_1, P_2, \ldots, P_n$. Antes de entrar em uma seção crítica, um processo P_i solicita um subconjunto de recursos existentes. P_i não pode continuar sua execução sem antes obter todos os recursos que ele requisitou, evitando com isso, o problema de *deadlock*. Na saída da seção crítica, esses recursos são liberados para outros processos.

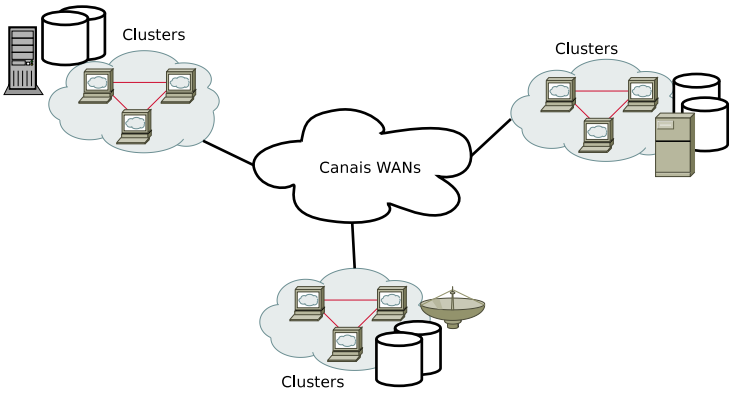

Figura 1.1: Arquitetura *Grid*

Diversos algoritmos baseados em *token* têm sido propostos para resolver o problema de exclusão mútua em ambientes distribuídos. Alguns deles consideram uma topologia hierárquica que reúne nós em grupos. Por exemplo, em [26] e [14] é apresentado uma estratégia baseada em grupos priorizados, onde nós com a mesma prioridade são reunidos em um mesmo grupo. Ambos os trabalhos propõem um modelo híbrido, onde o algoritmo que é executado dentro do grupo é diferente do algoritmo executado entre os grupos. Em [19], é proposto um algoritmo baseado em anéis de *clusters*, onde cada nó em um anel representa um *cluster* de nós. Apesar de terem uma abordagem hierárquica, esses algoritmos não consideram diferentes latências na comunicação entre nós distantes. Em [8], os autores apresentam um algoritmo baseado em *token*, inspirados no trabalho proposto por Naimi e Trehel [38]. O algoritmo proposto por estes autores considera diferentes latências na comunicação entre *sites* distintos. No entanto, este trabalho é limitado a somente a um único recurso compartilhado. Nosso algoritmo usa algumas das estratégias hierárquicas propostas por Bertier *et al.* [8]. Além disso, nosso algoritmo é estendido para resolver o

problema do compartilhamento de múltiplos recursos com múltiplas instâncias cada.

É típico em *Grids*, existir vários tipos de recursos compartilhados, incluindo hardware tais como CPU, espaço em disco, canais de comunicação, e software tais como programas, arquivos e dados. Veja em [12] e [23] exemplos detalhados de compartilhamento de software e hardware em *Grids*. Neste cenário, Maddi [33] apresenta algoritmos baseados no trabalho introduzido por Raynal [46], que considera a multiplicidade de recursos, mas apresenta baixo desempenho em um ambiente de larga escala composto por muitos nós. Em [9] também é proposto um algoritmo distribuído para o problema de alocação dinâmica de recursos, porém, este não considera a disparidade de latência entre os nós espalhados no sistema. Nosso algoritmo proposto também emprega alguns conceitos introduzidos por Bouabdallah e Laforest [9].

Comparamos o algoritmo proposto com o algoritmo baseado em *token*, introduzido por [9], que é um algoritmo bastante conhecido para resolver o problema de alocação de M recursos, mas que não considera particularidades de ambientes *Grids*.

1.1 Organização da Dissertação

O restante deste trabalho está organizado como segue. No Capítulo 2 apresentamos os trabalhos relacionados com o problema e os algoritmos propostos que foram utilizados como base para esta pesquisa. No Capítulo 3 descrevemos o algoritmo proposto. No Capítulo 4 apresentamos a maneira empregada de medir o desempenho do algoritmo, utilizando relógio lógico. Os resultados experimentais são apresentados no Capítulo 5. Finalmente, no Capítulo 6 concluímos o trabalho.

Capítulo 2

Trabalhos Relacionados

2.1 Introdução

O problema da exclusão mútua é um problema utilizado na resolução de conflitos no acesso aos recursos compartilhados. É considerado um problema fundamental em Ciência da Computação e tem sido amplamente estudado durante muito tempo.

Em sistemas distribuídos, vários processos executam seus *jobs* comunicando-se com outros processos. Quando estes processos compartilham recursos, eles podem eventualmente requerer acesso a determinados recursos ao mesmo tempo. Caso esses recursos precisem ser acessados de forma mutuamente exclusiva, é necessário que haja uma ordem de acesso a eles, pois somente um processo pode executar sua seção crítica usando os recursos alocados por vez [28].

Normalmente, sistemas distribuídos podem possuir uma grande quantidade de recursos distintos e dispersos pelos nós. Além disso, muitos usuários e aplicações podem precisar ter acesso a esses recursos em um mesmo tempo. Por esta razão, é necessário que os processos de diferentes nós cooperem entre si e se organizem para que eles possam acessar esses recursos de forma justa e consistente, evitando assim problemas como *starvation* e *deadlock* [49].

Diversos trabalhos que tratam do problema da exclusão mútua em sistemas distribuídos foram propostos na literatura. Muitos destes trabalhos utilizam diferentes técnicas para resolver o problema, além disso, algumas variações do problema também foram propostas. Grande parte dos algoritmos propostos na literatura se limita a uma variação do problema onde apenas uma única instância de recurso é compartilhada pelos processos, e

somente um processo pode acessá-la por vez, por exemplo, [48, 38, 61]. Porém, existem casos onde mais de um recurso existe no sistema distribuído. Assim, alguns trabalhos consideram a existência de k cópias idênticas de um único recurso compartilhado. Isto permite que um número limitado de processos executem suas seções críticas simultaneamente, cada um com uma única instância de cada recurso. Esse problema é conhecido como *k-mutual exclusion*, *k-mutex* ou então múltiplas entradas à seção crítica. Alguns exemplos podem ser vistos em [44, 36, 60, 56, 10].

Considerando a multiplicidade e a diversidade de recursos, uma outra variação desse problema, definida como *k-out of-M* recursos, permite que um processo solicite um número k de cópias tendo M recursos idênticos disponíveis, onde $1 \leq k \leq M$. Este problema foi resolvido por [46] e [33]. Por fim, uma versão generalizada do problema consiste do compartilhamento de múltiplos recursos com múltiplas instâncias cada recurso, isto é, um conjunto de recursos $R = (r_1, k_1), (r_2, k_2), \ldots, (r_m, k_m)$, tal que para cada tipo de recurso r_i existam k_i instâncias equivalentes. Em qualquer momento um processo pode necessitar de várias instâncias de diferentes tipos de recursos para entrar em sua seção crítica [46, 33, 9].

Na seção seguinte é descrita uma classificação básica utilizada pela maioria dos autores na literatura.

2.2 Classificação dos Algoritmos Distribuídos para Exclusão Mútua

Os algoritmos para exclusão mútua disponíveis na literatura, basicamente, podem ser divididos em duas principais classes: baseados em *token* e baseados em permissão (segundo [47]) ou não-baseados em *token* (segundo [52]). Ainda segundo [52], estes algoritmos também podem ser estáticos ou dinâmicos. Algoritmos são estáticos quando não armazenam informação sobre o estado atual do sistema e nem do histórico de execução da seção crítica, por outro lado, algoritmos dinâmicos podem tomar certas decisões com base na informação coletada ao longo de sua execução.

Diversos algoritmos propostos focam no aumento de desempenho, de acordo com uma certa métrica de desempenho, como por exemplo tempo de resposta e número de mensagens trocadas. Baseado na técnica utilizada, estes algoritmos podem ser classificados diferentemente. Na Figura 2.1, é mostrada uma classificação mais ampla dos algoritmos distribuídos para exclusão mútua desenvolvidos nas últimas décadas. Essa figura foi

extraída e remontada a partir dos trabalhos de [49, 52, 13].

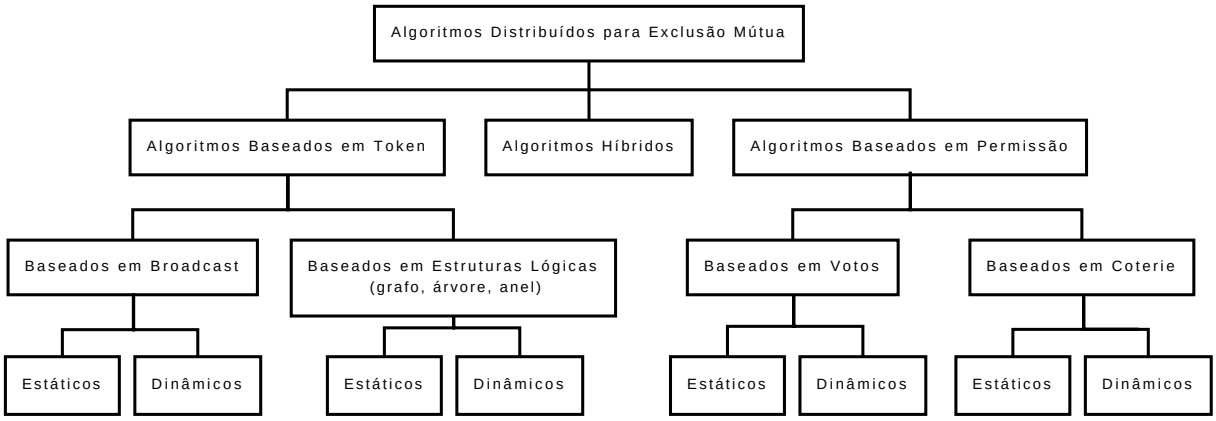

Figura 2.1: Árvore de classificação dos algoritmos distribuídos para exclusão mútua [52, 13]

2.2.1 Algoritmos Baseados em Permissão

Em algoritmos baseados em permissão, geralmente são necessárias sucessivas trocas de mensagens entre os processos para obter a permissão de executar a seção critica. Quando um processo necessita executar sua seção crítica, ele envia requisições pedindo permissão a todos os outros processos. O processo que recebe essa mensagem, caso não tenha interesse na seção crítica, concede sua permissão. Caso contrário, a decisão é obtida em favor do processo de maior prioridade, sendo esta feita utilizando o *timestamp* local deste processo. Existem muitos trabalhos na literatura que propõem resolver o problema da exclusão mútua usando o modelo baseado em permissão, alguns exemplos podem ser vistos em [48, 46, 29, 43].

Basicamente algoritmos baseados em permissão podem ser divididos em duas subcategorias, baseados em votos e baseados em *coterie*.

2.2.1.1 Baseados em Votos

O modelo baseado em votos consiste de um sistema de votação, onde cada processo tem direito a um voto (não negativo). Um processo recebe permissão para acesso à seção crítica, se ele obtiver no mínimo a maioria dos votos do total de votos realizados no sistema. Alguns algoritmos que utilizam esta técnica podem ser vistos em [58, 22, 4].

2.2.1.2 Baseados em *Coterie*

Nos algoritmos baseados em *coterie* como em [48], [34] e [41], utiliza-se o conceito de uma *coterie*, que seria uma coleção de conjuntos de processos do sistema. Para a obtenção do acesso à seção crítica, um processo necessita de permissão de cada processo de um conjunto pertencente a uma *coterie*.

Além dessa classificação, cada uma dessas duas categorias também pode ser dividida nos modelos estáticos (tais como [58, 22, 21, 48, 34]) e dinâmicos (tais como [5, 27, 51]).

Um estudo mais aprofundado sobre algoritmos baseados em permissão pode ser encontrado em [49] e [13].

2.2.2 Algoritmos Baseados em *Token*

Para se conseguir exclusão mútua, algoritmos baseados em *token* utilizam uma mensagem chamada de *token*, a qual é compartilhada pelos nós. Um nó só pode entrar em sua seção crítica após obter a posse do *token* que está relacionado ao recurso que ele necessita. Como o *token* é único, a exclusão mútua é então garantida.

Algoritmos baseados em *token* podem ainda ser classificados em baseados em *broadcasts* ou baseados em estruturas lógicas, como árvores, grafos e anéis. Esses algoritmos também podem ser estáticos ou dinâmicos [13, 52].

2.2.2.1 Baseados em Broadcast

Em algoritmos baseados em *broadcast*, nenhuma estrutura é imposta nos nós e um nó envia mensagens requisitando o *token* para outros nós em paralelo [52]. Os algoritmos estáticos são caracterizados por não possuírem memória a respeito de execuções na seção crítica. Nesses algoritmos, um pedido de *token* é enviado para todos os nós do sistema distribuído. Alguns exemplos podem ser vistos em [42, 57]. Por outro lado, algoritmos dinâmicos como [50] e [61], são capazes de se lembrar do histórico da localização do *token* e enviam mensagens pedindo *token* somente para nós dinamicamente selecionados, os quais provavelmente estarão com o *token*.

Em relação a desempenho, algoritmos baseados em *broadcast* geralmente possuem um alto tráfego de mensagens. Algoritmos estáticos requerem N mensagens por cada execução à seção crítica [57], enquanto algoritmos dinâmicos requerem $N/2$ em baixa carga, chegando a N em alta carga, de acordo com [50].

2.2.2.2 Baseados em Estruturas Lógicas

Em algoritmos baseados em estruturas lógicas, nós são organizados em uma configuração lógica [52]. Essas estruturas basicamente podem ser baseadas em árvore, grafo ou em anel. Estes algoritmos também podem ser estáticos ou dinâmicos.

Baseados em Árvore: Em algoritmos baseados em árvore, nós normalmente são organizados de forma que os pedidos sejam propagados seqüencialmente através dos caminhos entre o nó que requisita e o nó que mantém o *token*, e da mesma forma também para a passagem do *token* solicitado.

Nos algoritmos estáticos baseados em árvore [45, 40], a estrutura lógica se mantém imutável e somente a direção das arestas é modificada durante a passagem dos pedidos de *token* pelos caminhos que interligam os nós até a raiz da árvore.

Os algoritmos dinâmicos como, por exemplo, [37], [16] e [38], mantém-se uma árvore dinâmica, tal que a raiz dessa árvore é sempre o último nó a obter o *token* dentre os pedidos correntes feitos por todos os nós. A estrutura da árvore é modificada dinamicamente à medida que uma requisição de *token* é enviada pelos nós da árvore até a raiz. Cada nó intermediário por onde o pedido foi passado, passa a considerar o nó requisitante como um nó "pai" ou o nó "provável dono do *token*", exceto quando o nó intermediário é a raiz da árvore. Neste caso, o nó requisitante se torna a nova raiz dessa árvore e os pedidos posteriores são enviados até ele. Quando um pedido chega até o nó raiz, este envia o *token* diretamente ao próximo nó que requisitou para executar sua seção crítica.

Algoritmos baseados em árvores e em grafos geram um baixo tráfego de mensagens. Uma estudo extensivo de medição feito por Raymond [45], mostrou que a construção randômica de uma árvore composta de N nós, tipicamente possui uma complexidade média de $O(log(N))$. Além disso, a característica dinâmica da estrutura lógica baseada em árvore, que possibilita que ela seja modificada em tempo de execução e o baixo consumo de mensagens, permitem conseguir grandes vantagens no uso de estratégias específicas de ambientes com diferentes configurações, tais como ambientes com topologias hierárquicas [8].

Baseados em Grafos: Nas estruturas baseadas em grafos, nós são estruturados na forma de um grafo direcionado com um sumidouro (*sink*) o qual mantém o *token*. As requisições e a propagação do *token* são manipuladas da mesma forma que nos algoritmos

baseados em árvore, com a vantagem de serem tolerantes a falhas devido aos múltiplos caminhos entre os nós, enquanto na árvore não. No entanto, esta estrutura por ser tolerante a falhas possui um alto custo de mensagens que são usadas para prevenir os ciclos na estrutura do grafo. Veja [15, 24, 59] para mais detalhes.

Baseados em Anel: Em uma estrutura lógica usando anel, os nós são arranjados na forma de um anel lógico, onde o *token* circula pelo anel em uma direção fixa através de mensagens ponto a ponto. Nenhum pedido explícito é necessário, um nó que queira acessar sua seção crítica precisa apenas aguardar que o *token* chegue até ele. Ao sair da seção crítica, o nó redireciona o *token* ao seu sucessor no anel. Exemplos de algoritmos baseados em anéis podem ser vistos em [31], [11] e [33].

2.2.3 Algoritmos Híbridos

Embora vários algoritmos distribuídos para exclusão mútua tenham sido propostos para minimizar tanto o tráfego de mensagens quanto o atraso de tempo, nenhum deles pode minimizar ambos ao mesmo tempo [52, 13]. Por exemplo, o algoritmo de Maekawa [34] reduz o tráfego de mensagens para $O(\sqrt{N})$; porém, possui um grande atraso de tempo em sucessivas execuções na seção crítica quando comparado ao algoritmo de Ricart-Agrawala [48] que possui $O(N)$ [13]. Nesse contexto, alguns autores propuseram uma estratégia híbrida utilizando diferentes tipos de algoritmos, baseados em *token* e baseados em permissão. Os algoritmos híbridos para exclusão mútua têm sido utilizados no intuito de minimizar ambos, o atraso de tempo e a complexidade de mensagem [14]. Alguns exemplos podem ser encontrados em [14] [26].

2.3 Soluções Existentes para Ambientes Hierárquicos e de *Grids*

Ambientes hierárquicos, tais como *Grids*, geralmente são compostos por diversos processos agrupados em *clusters*, e que são dispersos geograficamente. Tais *clusters* são interconectados por canais remotos de comunicação, que geralmente são mais lentos do que a comunicação local feita por processos dentro de cada *cluster*.

Neste cenário hierárquico, Housni *et al.* [26] e Chang *et al.* [14] propuseram soluções utilizando o conceito de grupos priorizados, onde os processos, seguindo algum critério, são arranjados em grupos. Assim, é possível priorizar a comunicação local com relação

à comunicação global. Além disso, eles também utilizam a estratégia de algoritmos híbridos, onde o algoritmo de exclusão mútua executado dentro do grupo é diferente do algoritmo executado entre os grupos. Apesar de possuírem uma abordagem semelhantemente hierárquica, estes trabalhos são limitados, pois compartilham apenas uma única instância de recurso. Além disso, a estratégia hierárquica proposta nesses trabalhos ainda não é adequada, pois a maior latência nos canais de comunicação que interconectam os nós distantes não é levada em conta. Outra proposta semelhante a estas, é feita por Erciyes [19]. O autor utiliza um modelo baseado em anéis de *clusters*, onde cada nó de um anel representa um *cluster* de nós. Assim como no caso anterior, esta proposta também não considera múltiplos recursos, e também não considera a maior latência nos canais de comunicação que interlingam os nós remotos.

Em Bertier *et al.* [8], os autores apresentam um algoritmo baseado em *token*, inspirados no trabalho proposto por Naimi e Trehel [38]. O algoritmo proposto por estes autores considera a diferença de latências na comunicação entre nós distintos e apresenta estratégias para otimizar a troca de mensagens e a passagem do *token* pelo sistema considerando a hierarquia. No entanto, este trabalho é limitado a somente um recurso compartilhado. Este trabalho esta melhor discutido na Subseção 2.4.1.1.

Mais recentemente, em Sopena *et al.* [55], os autores propuseram um mecanismo que considera a heterogeneidade da latência na comunicação em ambientes de *Grid*. Este algoritmo permite a execução de algoritmos de exclusão mútua diferentes, tanto entre processos dentro do *cluster* quanto entre os *clusters*. Esta estratégia permite avaliar o comportamento de diferentes composições de algoritmos. Porém, este trabalho também não leva em consideração a existência dos múltiplos recursos de ambientes de *Grid*.

2.4 Soluções Relacionadas à Proposta

Nesta seção são descritos separadamente alguns dos principais trabalhos que nortearam a construção do algoritmo proposto. Inicialmente, são descritas as soluções hierárquicas propostas pelo trabalho de [7, 8], bem como, as soluções que contemplam o problema da exclusão mútua para múltiplos recursos [9], os quais inspiraram o desenvolvimento do algoritmo proposto.

2.4.1 Soluções Hierárquicas

2.4.1.1 Algoritmo de `Bertier et al.`

Bertier *et al.* [7, 8] apresentaram adaptações do algoritmo baseado em *token* proposto por Naimi e Trehel [38] e [36], explorando a topologia hierárquica de *Grid*. Assim como no trabalho de Naimi-Trehel, no algoritmo proposto em Bertier *et al.*[8] os autores também consideram um ambiente onde exista apenas um recurso compartilhado, sendo este representado por um *token*.

A estrutura lógica dinâmica herdada de Naimi e Trehel consiste de uma árvore composta pelos processos do sistema. Esta estrutura lógica e dinâmica da árvore faz com que o processo que está na raiz seja sempre o último a receber o *token* pelas requisições correntes.

Por serem estruturas flexíveis, podendo ser modificadas dinamicamente durante a execução, e por possuírem um menor tráfego de mensagens comparado a mecanismos como o de *broadcast*, as estruturas baseadas em árvores escalam melhor, portanto são mais adequadas para aplicações em *Grid*.

Os algoritmos adaptados pelos autores levam em consideração algumas características importantes encontradas em *Grids*, como por exemplo, as diferenças de latências nos canais que interligam processos de diferentes *clusters*.

Em seu trabalho, Bertier *et al.* propuseram algumas versões do algoritmo de Naimi-Trehel. Em uma delas é explorada a localidade dos processos, priorizando pedidos de *tokens* feitos dentro do *cluster*, localmente, em relação aos pedidos feitos por processos de *clusters* remotos. Esse procedimento garante uma melhor utilização dos canais remotos, evitando o uso arbitrário desses canais.

Os autores também modificaram a forma de inicialização do algoritmo original de Naimi-Trehel, fazendo com que os processos inicialmente apontassem para um processo líder em cada *cluster*. Assim, no início da execução do algoritmo, os pedidos de recursos são direcionados a esses líderes. Desta forma, consegue-se um balanceamento do tráfego de mensagens iniciais, pois ao invés de todos os processos do sistema enviarem pedidos à apenas um processo, eles enviam esses pedidos aos líderes de seus *clusters*.

2.4.2 Soluções para Múltiplos Recursos

2.4.2.1 Algoritmo de `Bouabdallah e Laforest`

O problema da alocação de múltiplos recursos aparece quando um processo necessita de executar sua seção crítica não apenas com um único recurso, e sim com um subconjunto qualquer de recursos distintos disponíveis. Neste contexto, Bouabdallah e Laforest [9] propuseram um algoritmo distribuído baseado em *token* para alocação de múltiplos recursos. Neste trabalho, os autores consideram um ambiente onde não exista diferença de latências entre os canais de comunicação do sistema distribuído, tratando todos os processos igualmente no sistema. O algoritmo também utiliza uma estrutura lógica de árvore, baseada no algoritmo proposto por Naimi e Trehel [38] e [36], o que permitiu uma melhor integração das estratégias utilizadas no algoritmo proposto.

Uma característica que o diferencia da maioria das propostas é a capacidade de alocar vários recursos simultaneamente, respeitando o acesso exclusivo. Cada instância de recurso disponível no sistema é associada a um *token*. Para que um processo tenha acesso a sua seção crítica, ele precisa possuir todos os *tokens* associados aos recursos que ele necessita. Para evitar que processos busquem pelos *tokens* individualmente, os autores propuseram uma estrutura lógica, denominada de *Control Token* (CT).

O *Control Token* exerce o papel de um *token* centralizador que concede acesso à seção crítica para quem o possuir. Além disso, o CT também armazena informações a respeito de cada *token* no sistema, essas informações permitem saber quais *tokens* estão disponíveis, e para o caso de estarem ocupados, permite saber quem é o processo provável dono daquele *token*.

Diferentemente da proposta de Naimi e Trehel, onde apenas um *token* é trocado entre os processos, em Bouabdallah e Laforest a estrutura do CT é trocada entre os processos. Somente um processo pode obter o CT de cada vez. Isso implica que somente um processo pode ler e escrever na estrutura lógica naquele tempo, garantindo assim a exclusão mútua. Um processo deve solicitar a posse do CT para que ele tenha acesso às informações a respeito dos *tokens* e possa enviar pedidos aos processos respectivos que possuem ou irão possuir cada *token* que ele necessita.

Os autores Bouabdallah e Laforest também idealizaram uma generalização do algoritmo, no qual considera não somente o compartilhamento de múltiplos recursos, mas também múltiplas instâncias para cada tipo de recurso. Esta generalização foi implementada para esta dissertação de mestrado e foi utilizada como base de comparação com o

algoritmo proposto.

2.5 Soluções Recentes

Além dos trabalhos que utilizam o conceito de algoritmos híbridos para exclusão mútua e algoritmos que consideram as diferenças de latências nos canais de comunicação, ou ainda, algoritmos que consideram a multiplicidade de recursos, existem também alguns trabalhos que focam em um conceito ainda não suportado pelo algoritmo proposto, que é a tolerância a falhas. Neste cenário, alguns trabalhos podem ser citados por possuírem um alto grau de relevância e proximidade à proposta deste trabalho, como um próximo passo a ser desenvolvido, como por exemplo [3, 54, 53].

Capítulo 3

O Algoritmo Proposto

3.1 Introdução

O ambiente distribuído considerado nesta proposta consiste de n processos com um identificador único cada, que se comunicam de forma assíncrona por troca de mensagens. Mensagens são entregues em um tempo finito, porém imprevisível. Assume-se também que falhas não ocorrem.

O algoritmo proposto utiliza o modelo baseado em *token*, onde cada *token* representa uma instância de recurso disponível no sistema. Uma condição necessária para um processo entrar em sua seção crítica (SC) é possuir todos os *tokens* referentes aos recursos dos quais ele necessita. O tempo de execução de cada seção crítica é finito, porém não limitado.

A solução para este problema precisa garantir a propriedade de exclusão mútua, ou seja, uma instância de recurso pode somente ser utilizada por no máximo um processo por vez. É necessário também garantir que não ocorra o problema de *starvation*, qualquer processo que solicitar recursos irá obtê-los em um tempo finito. Além disso, deve se garantir a ausência de *deadlocks*.

Considerando o problema da alocação de múltiplos recursos, o algoritmo proposto é baseado não somente em *tokens* que garantem acessos aos recursos, mas também em uma estrutura única denominada de *Control Token* (CT) que ordena os acessos aos recursos. Essa estrutura foi originalmente proposta por [9], no entanto foi adaptada neste trabalho considerando seu uso em ambientes hierárquicos, tais como *Grid*. O CT é uma estrutura lógica que mantém os *tokens* livres e também uma visão para cada *token* não livre, do

processo que o possui ou que irá possuí-lo em um tempo finito. Um único processo por vez mantém o *Control Token* que consiste de:

- um conjunto A de *tokens* livres;
- uma lista B de processos, cada um com o conjunto de *tokens* que ele possui ou possuirá;
- uma lista C de recursos, cada um com um processo que possui prioridade de acesso a esse recurso e um valor associado, representando sua freqüência de uso pelo processo.

Quando um processo recebe o CT, ele pega todos os *tokens* que ele necessita no conjunto A, adiciona em A os *tokens* desnecessários que ele possui e atualiza a lista B indicando que ele possui ou possuirá todos os *tokens* que ele pegou. Esta última ação causa o envio de requisições de recursos aos processos que os possuem. A lista C é usada para escolher os processos mais apropriados da lista B para enviar tais requisições. Ao receber todas as respostas correspondentes, o CT pode então ser liberado para um próximo processo.

Este algoritmo tem como objetivo a redução de mensagens entre *clusters* diferentes e na média do tempo esperado na obtenção dos recursos. Para reduzir mensagens remotas, um processo líder é eleito em cada *cluster* para que concentre os pedidos iniciais de CT, que eventualmente pode estar em outro *cluster*. Além disso, os pedidos locais de CT, ou seja, pedidos feitos por processos que estão no mesmo *cluster*, são priorizados.

Em relação aos *tokens* que representam as instâncias de recursos, o algoritmo também prioriza sua transmissão aos processos de um mesmo *cluster*, o que permite ter uma maior concentração de mensagens locais, reduzindo assim, a troca de mensagens remotas. É definido também, um mecanismo de prioridades para processos em cada recurso. Assim, pedidos de recursos são redirecionados aos processos que os utilizam com menos freqüência.

O algoritmo satisfaz o requisito de exclusão mútua, visto que um único *token* é associado a uma única instância de recurso e somente um processo pode obtê-lo a cada momento. Ele também é livre de *starvation*, pois cada processo após realizar um pedido de CT irá recebê-lo em um espaço finito de tempo e, também, porque cada processo que necessite dos *tokens* para executar sua seção crítica irá obtê-los também em um tempo finito. As técnicas utilizadas no algoritmo proposto foram inspiradas nos algoritmos de [9] e [8] e suas análises, também, são válidas para essa proposta.

As seções seguintes descrevem estes procedimentos em detalhes.

3.2 Descrição do algoritmo

A estrutura lógica de árvore utilizada é baseada em um modelo proposto por [38], a qual é construída de forma em que cada processo no sistema, exceto aquele que possui o CT, aponte para um processo que de seu conhecimento possui ou possuirá o CT, veja um exemplo na Figura 3.1. Essa informação é armazenada em uma variável "*owner*", que é atualizada dinamicamente todas as vezes que um processo recebe uma requisição pedindo o CT. Em cada momento, o último processo a pedir o CT é sempre a raiz da árvore lógica, e quando isso acontece o conteúdo de sua variável *owner* recebe vazio. Quando o pedido de um processo chega até o processo raiz da árvore, ele é adicionado em uma fila distribuída definida pela variável "*next*", que mantém a ordem na qual o CT será passado pelos processos que o solicitaram. Os elementos ilustrados na Figura 3.1 são detalhados na Subseção 3.2.1.

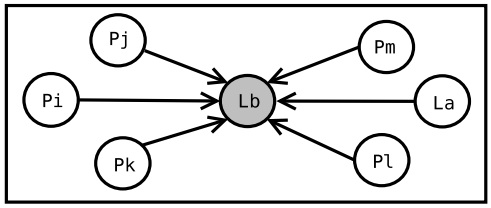

Figura 3.1: Configuração lógica de árvore tradicional, segundo [38].

3.2.1 Configuração Inicial da Árvore

Baseado na idéia proposta por [8], foi feita uma modificação na configuração inicial desta árvore como pode ser visto na Figura 3.2. A figura representa um ambiente hierárquico composto por dois *clusters*, *cluster* A e *cluster* B, onde P_j, P_i, P_k e L_a pertencem ao *cluster A* e P_l, P_m e L_b pertencem ao *cluster B*.

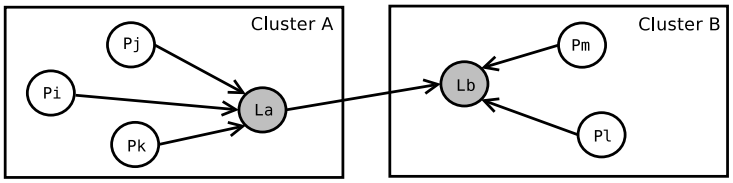

Figura 3.2: Configuração lógica de uma árvore com hierarquia

Em cada *cluster*, um processo é escolhido como líder para o qual é enviada a primeira requisição de CT de cada processo daquele *cluster*, no exemplo, L_a e L_b são os líderes. Processos líderes eventualmente também acessam suas seções críticas. A configuração dessa estrutura lógica inicial visa concentrar as requisições iniciais localmente nos *clusters*,

evitando o envio de mensagens remotas. Veja também no Algoritmo 1, o procedimento de inicialização do algoritmo e o processo de configuração inicial da árvore. Observe que um processo é eleito como mantenedor inicial do CT, além disso, em cada *cluster* é definido um processo líder e todos os outros processos apontam para ele na árvore. As variáveis contidas nesse algoritmo são descritas no decorrer deste capítulo.

Algoritmo 1 - Inicialização

```
1: requesting ← FALSO
2: next ← 0
3: remote_owner ← 0
4: num_preempt ← 0
5: se NO_ELEITO ∈ Processo_Locais então
6:    se self = NO_ELEITO então
7:       control_token ← VERDADE
8:       owner ← 0
9:    senão
10:      control_token ← FALSO
11:      owner ← NO_ELEITO
12:   fim se
13: senão
14:   control_token ← FALSO
15:   se self = Lider_i então
16:      owner ← NO_ELEITO
17:   senão
18:      owner ← Lider_i
19:   fim se
20: fim se
```

O procedimento de obtenção do acesso à seção crítica se divide em dois passos. Primeiro, é necessário obter o CT que garante o direito ao acesso exclusivo à seção crítica, além de obter as informações necessárias para o segundo passo, que é obter os *tokens* necessários. Esses procedimentos e os detalhes do algoritmo proposto são mostrados separadamente nas subseções seguintes.

3.2.2 Estágio 1 - Obtenção do *Control Token*

Neste estágio, basicamente são utilizadas três tipos de mensagens, *Requisicao*, *ControlToken* e *Preempcao*, as quais são tratadas por procedimentos específicos no processo receptor, que são descritos posteriormente nesta subseção.

Cada processo possui uma lista de *tokens* disponíveis localmente, ou seja, os tokens que somente este processo mantém, no momento. Quando um processo P_i necessita de um conjunto de recursos para entrar em sua seção crítica, ele primeiro verifica se os *tokens* correspondentes estão disponíveis localmente. Se sim, ele trava os *tokens* e executa sua seção crítica. Do contrário, ele solicita o CT.

A estrutura de dados utilizada na implementação do CT é um vetor de k posições, onde k é o número total de *tokens* que representam as instâncias dos tipos de recursos disponíveis no sistema. Nessa estrutura de dados, os índices k representam os *tokens* e suas posições guardam o identificador do processo que possui àquele *token* ou um valor vazio, indicando que o *token* está livre. Além disso, uma lista auxiliar *prioridades*, é acoplada à estrutura de dados para guardar as informações referentes às prioridades dos processos em cada tipo de recurso.

No início o *Control Token* é alocado para um dos líderes, que é definido pela constante NO_ELEITO, e os pedidos de outros líderes são redirecionados a ele. Cada líder mantém a identificação do nó que realizou a última requisição local, evitando transmissões de mensagens entre *clusters*. Há aqui dois casos a serem considerados que compõem os exemplos a seguir.

3.2.2.1 Exemplo de requisição local de CT

Neste primeiro exemplo, considere um *cluster A* que não possui o CT e onde um processo P_i envia uma requisição de CT para o processo L_a. Se um outro processo local P_j anteriormente enviou uma requisição de CT e L_a está ciente disso, L_a então redireciona a requisição de P_i para P_j evitando a transmissão de uma mensagem para um *cluster* remoto. Note que a requisição original de P_j fez com que L_a redirecionasse aquela mensagem para o líder do *cluster* que possui o CT. Observe a ordem na qual as requisições foram enviadas na Figura 3.3. Quando o CT chega ao *cluster A*, ele segue o caminho mostrado na Figura 3.4.

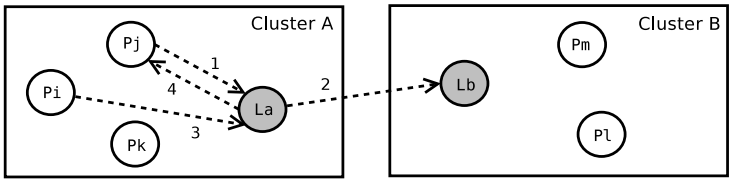

Figura 3.3: Requisições de *Control Token*

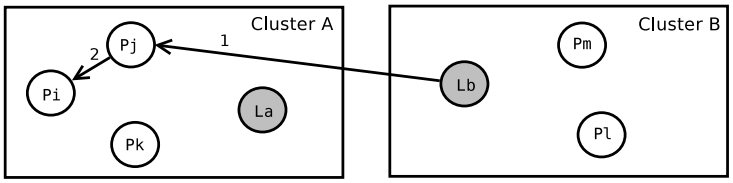

Figura 3.4: Caminho 1 do *Control Token*

Os detalhes do procedimento referente a uma requisição de CT podem ser vistos no

Algoritmo 2. Seguindo o exemplo acima, a requisição feita pelo processo P_i ao processo L_a ilustrada na Figura 3.3, foi feita executando o procedimento *Requisicao_SC* do algoritmo. Neste procedimento, inicialmente é verificado se o processo está requisitando a seção crítica e em seguida se ele já possui o CT. Caso ele não o possua, uma mensagem requisitando o CT é enviada ao processo indicado em sua variável local *owner*, que armazena o provável dono do CT. Após isso, ele fica em estado de espera aguardando o recebimento do CT. Note que nesse momento P_i se tornou a atual raiz da árvore, atualizando sua variável *owner* com 0. Da mesma forma, L_a ao receber o pedido de P_i o indicou em sua variável *owner*, com isso, os próximos pedidos de CT devem ser enviados a P_i.

Algoritmo 2 - Requisição para acesso à seção crítica

```
1: Requisicao_SC( )
2: requesting ← VERDADE
3: se control_token = FALSO então
4:    Envia ⟨Requisicao, S_i⟩ para owner
5:    owner ← 0
6:    Espera receber a mensagem ⟨ControlToken⟩
7: fim se
8: { Parte suprimida }
```

O Algoritmo 3, apresenta uma versão simplificada da ação executada no recebimento de uma requisição, que no nosso exemplo é executada pelo processo local L_a. Observe entre as linhas 2 e 5 que se caso o processo fosse a raiz da árvore, ele a modificaria indicando P_i como a atual raiz, atualizando a variável *owner*. Além disso P_i é adicionado a uma fila distribuída representada pela variável *next*, que ordena o recebimento do CT. Se caso o CT estivesse livre em L_a, ele poderia ser enviado imediatamente a P_i (linhas 6 a 10). Porém, de acordo com o exemplo iniciado na Figura 3.3, o trecho descrito entre as linhas 11 e 14 é executado e L_a redireciona a requisição de P_i ao processo P_j que de acordo com o conteúdo de *owner* é o próximo a possuir o CT.

3.2.2.2 Exemplo de requisição remota de CT

No segundo caso, tem-se um cenário similar, porém com o CT no *cluster A*. Considere agora que L_a receba uma requisição remota originada do processo P_l de um *cluster B*, através do processo L_b. Então, L_a redireciona sua requisição para o último processo do *cluster* no qual tenha requisitado o CT, no exemplo (Figura 3.3) P_i, mas continua a considerar P_i como o último processo a ter requisitado o CT. Ao receber a requisição de P_l pedindo o CT, o processo P_i considera ele como o próximo a receber o CT.

Porém, caso o processo P_i receba uma requisição de um processo local P_k, ele insere a requisição anterior do processo P_l em uma fila de processos remotos, priorizando a

Algoritmo 3 - Recebe pedido de CT (versão simplificada)

```
1: Recebe_Requisicao(S_j):
2: se owner = 0 então
3:    se requesting = VERDADE então
4:       next ← S_j
5:       owner ← S_j
6:    senão
7:       Envia ⟨ControlToken, CT⟩ para S_j
8:       owner ← S_j
9:       control_token ← FALSO
10:   fim se
11: senão
12:   Envia ⟨Requisicao, S_j⟩ para owner
13:   owner = S_j
14: fim se
15:
```

requisição local de P_k. Assim, ele considera P_k como o próximo candidato a receber o CT e em seguida, então, envia essa fila para P_k, que passa a considerar P_l como o próximo a receber o CT. Essa estratégia, a qual denominamos de preempção foi adotada com o objetivo de priorizar os atendimentos locais, evitando que o CT fique trafegando entre os *clusters*, através de canais remotos. Observe na Figura 3.5 o caminho a ser seguido pelas mensagens do CT. Para evitar *starvation*, é utilizado um mecanismo que limita o número de preempções realizadas, ou seja, o número de vezes que o envio do CT a um processo remoto é postergado dando lugar a um processo local. Esse valor é definido como um parâmetro do algoritmo e pode ser configurado de acordo com a aplicação.

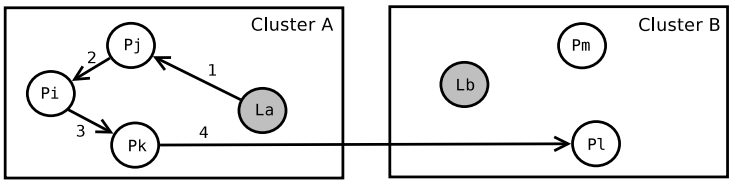

Figura 3.5: Caminho 2 do *Control Token*

Para descrever o exemplo de execução, é usada uma versão estendida da ação *Recebe_Requisicao* que pode ser vista no Algoritmo 4. As principais modificações nessa função são: o procedimento que diferencia processos locais de processos remotos e o procedimento que realiza preempções. De acordo com o exemplo, há uma ocorrência de uma requisição remota originada do processo P_l, que primeiramente foi enviada ao processo L_b líder do *cluster B*, e então redirecionada ao líder do *cluster A*, no caso L_a. Observe que executando o trecho das linhas 27 a 31 do Algoritmo 4, L_a ao receber uma requisição remota, redireciona esta requisição a P_i, e continua mantendo a identificação de P_i em sua variável *owner*. Quando esta mensagem, que foi redirecionada por L_a, chega a P_i, ele então adiciona P_l como o próximo da fila a receber o CT e armazena aquele processo em

uma variável *remote_owner*, indicando que um processo remoto é o último a obter o CT até o momento. Esse procedimento pode ser visto entre as linhas 4 e 10 do Algoritmo 4.

No entanto, em um outro momento, P_i recebe uma requisição originada de um processo local P_k, através de um redirecionamento feito por L_a. Neste caso, o processo P_i então executa o trecho compreendido pelas linhas 11 a 17 do Algoritmo 4, onde P_i realiza uma preempção, priorizando a requisição local de P_k, adicionando-o antes de P_l. Para isso, uma mensagem de preempção é enviada ao processo P_k, contendo a identificação do processo remoto que está enfileirado e o número total de preempções realizadas. O número total de preempções não deve ultrapassar o limite pré-estabelecido pela variável $MAX\_PREEMPT$, que é um parâmetro do algoritmo. Veja na linha 12 do Algoritmo 4 onde é feita esta verificação. Caso este limite seja alcançado, o processo remoto então é atendido, evitando assim que um processo remoto fique esperando infinitamente, o que seria um problema de *starvation*.

Algoritmo 4 - Recebe pedido de CT (versão estendida)

```
 1: Recebe_Requisicao(S_j):
 2: se owner = 0 então
 3:    se requesting = VERDADE então
 4:       se next = 0 então
 5:          next ← S_j
 6:          se S_j ∈ Processos_Locais então
 7:             owner ← S_j
 8:          senão
 9:             remote_owner ← S_j
10:          fim se
11:       senão
12:          se S_j ∈ Processos_Locais e num_preempt < MAX_PREEMPT então
13:             num_preempt ← num_preempt + 1
14:             owner ← S_j
15:             Envia ⟨Preempcao, num_preempt, next⟩ para S_j
16:             next ← S_j
17:          senão
18:             Envia ⟨Requisicao, S_j⟩ para next
19:             owner ← S_j
20:          fim se
21:       fim se
22:    senão
23:       Envia ⟨ControlToken, CT, num_preempt⟩ para S_j
24:       owner ← S_j
25:       control_token ← FALSO
26:    fim se
27: senão
28:    Envia ⟨Requisicao, S_j⟩ para owner
29:    se S_j ∈ Processos_Locais então
30:       owner = S_j
31:    fim se
32: fim se
```

O procedimento que trata o recebimento de uma mensagem de preempção é mostrado

no Algoritmo 5. Note que, primeiramente, o contador do número de preempções realizadas é atualizado, e logo em seguida é propagado (linha 7) até chegar ao último processo que recebe o CT localmente (linhas 3 a 5), sendo este o responsável por enviar o CT ao processo remoto em questão.

Algoritmo 5 - Recebe mensagem de preempção

```
1: Recebe_Preempcao(num_preempt_j, remote_node):
2: num_preempt ← num_preempt + num_preempt_j
3: se next = 0 então
4:     next ← remote_node
5:     remote_owner ← remote_node
6: senão
7:     Envia ⟨Preempcao, num_preempt, remote_node⟩ para owner
8: fim se
9:
```

No Algoritmo 6 é possível ver o procedimento responsável pelo evento que trata o recebimento do *Control Token*. Além da estrutura do CT que é passada na mensagem, é enviado também o número total de preempções realizadas até o momento, para que este processo atualize seu contador local.

Algoritmo 6 - Recebe control *token*

```
1: Recebe_ControlToken(CT_j, num_preempt_j):
2: num_preempt ← num_preempt + num_preempt_j
3: CT ← CT_j
4: control_token ← VERDADE
5:
```

Note que à medida que o algoritmo progride, as próximas requisições de CT de cada processo são enviadas ao processo que de seu conhecimento é o provável dono do CT. Em ambos os casos previamente descritos, uma próxima requisição de P_j iria ser enviada a P_i. Mais detalhes deste procedimento podem ser encontrados em [8].

3.2.3 Estágio 2 - Obtendo *Tokens* dos Recursos

Durante a execução do algoritmo, um *token* pode estar tanto no CT quanto na posse de algum processo. Neste último caso, o *token* pode ser travado, quando o processo está utilizando a instância de recurso correspondente, ou pode estar livre.

Neste estágio, primeiramente um processo tenta encontrar todos os *tokens* necessários no conjunto A do *Control Token*. Para escolher os *tokens* do conjunto A, um processo pode considerar diversos critérios, tais como localização física dos recursos que os *tokens* representam ou suas características de execução. Existem diversos trabalhos na literatura

que abordam o problema de seleção de recursos para aplicações *Grid*, veja exemplos em [32] e [39].

A Figura 3.6 mostra a representação lógica dos *clusters* e logo acima a estrutura lógica do *Control Token* recebida pelo processo P_i. Nessa figura o CT é ilustrado por um conjunto de tipos de recursos, onde cada tipo de recurso é composto por um conjunto de *tokens* que representam as instâncias dos recursos e os processos que eventualmente os mantêm. Também são mostradas as informações referentes aos *tokens* necessários para um processo acessar sua seção crítica e os *tokens* que ele possui localmente. Para o exemplo, considere hipoteticamente que o processo necessite de duas instâncias do recurso R_1, uma instância de R_2 e uma de R_4 para acessar sua seção crítica. Considere também que ele possua uma instância do recurso R_3, no caso $T6$, que ele não irá utilizar.

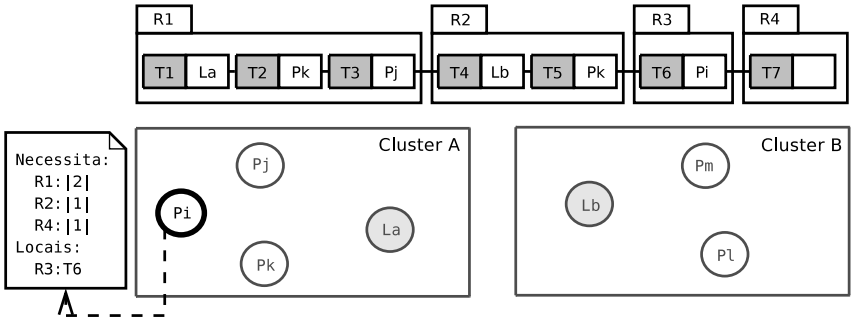

Figura 3.6: Estrutura do *Control Token* recebida

Como existe uma instância livre do recurso R_4, a qual o processo P_i necessita, ele neste caso pode obtê-la e atualizar o *Control Token* informando que ele agora a possui. Veja na Figura 3.7 que ilustra esse procedimento. Observe também, através do Algoritmo 7, que após obter os *tokens* livres do *Control Token* (linhas 2 a 6), caso o processo encontre todos os recursos dos quais ele necessita, ele poderá então acessar exclusivamente sua seção crítica (linhas 12 a 15). Do contrário, ele invoca um procedimento para seleção dos recursos dos quais ele irá pedir aos processos que os mantêm.

Outro procedimento utilizado, é o de devolver instâncias não utilizadas ao *Control Token*, para que um outro processo que obtenha o CT e que necessite de tais *tokens*, evite mensagens desnecessárias de pedido. A Figura 3.8 mostra o momento quando o processo P_i devolve o *token* T_6 referente ao recurso R_3. No Algoritmo 7, linhas 7 a 9, também é mostrado como esse procedimento é realizado.

Quando um processo não encontra todos os *tokens* necessários no conjunto A, usando a lista B ele determina de quais processos ele irá requisitar os *tokens* indisponíveis. No algoritmo proposto, um processo pode escolher outro da lista B para enviar uma requisição

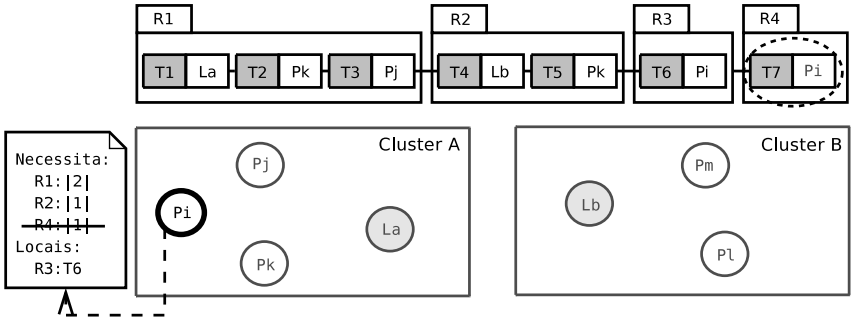

Figura 3.7: Obtem os *tokens* livres do CT

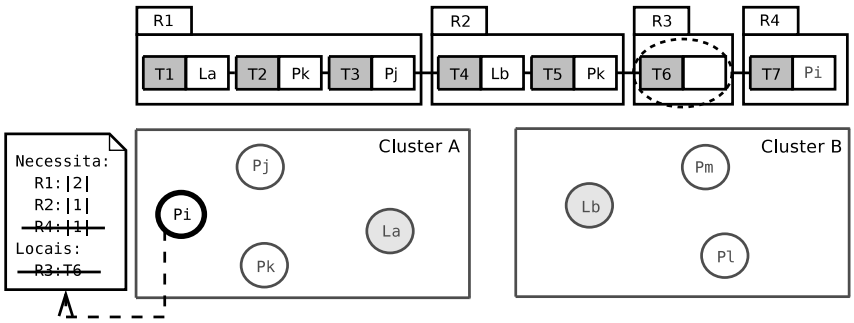

Figura 3.8: Libera *tokens* não utilizados ao CT

Algoritmo 7 - Pega os *tokens* livres no CT

```
1: Pega_Tokens_Livres():
2: para todo token t tal que 0 ≤ t < k faça
3:    se CT[t] = 0 e t ∈ tokens_necessarios então
4:       tokens_locais ← tokens_locais ∪ t
5:       CT[t] ← self
6:    fim se
7:    se t ∉ tokens_necessarios e t ∈ tokens_locais então
8:       tokens_locais ← tokens_locais − t
9:       CT_t ← 0
10:   fim se
11: fim para
12: trava ⟨tokens_locais⟩
13: se tokens_necessarios ⊆ tokens_locais então
14:    {Acessa a seção crítica}
15: senão
16:    Seleciona_Tokens ()
17: fim se
18:
```

de recurso obedecendo a seguinte ordem de prioridades:

1. um processo local que não usa freqüentemente o recurso;
2. um processo local que usa freqüentemente o recurso;
3. um processo remoto que não usa freqüentemente o recurso, e finalmente;

4. um processo remoto que usa freqüentemente o recurso.

Essa etapa pode ser vista no Algoritmo 8, onde estes critérios de seleção são tratados. Na função *Seleciona_Tokens*, o *Control Token* é percorrido para cada um dos critérios de seleção, selecionando para cada vez, um eventual conjunto de *tokens* e processos. Essa informação é armazenada em um conjunto temporário de pedidos, para que mais tarde, o processo possa enviar esses pedidos de *token* aos respectivos processos que os mantêm.

Algoritmo 8 - Seleciona os *tokens* necessários

```
1: Seleciona_Tokens():
2: Conjunto_Pedido ← ∅
3: Seja a tupla (S_j, T) tal que S_j ∈ CT e T ⊆ tokens_necessarios
4: para todo S_j ∈ Processos_Locais e S_j ∉ CT.prioridade faça
5:    Conjunto_Pedido ← Conjunto_Pedido ∪ (S_j, T)
6: fim para
7: para todo S_j ∈ Processos_Locais e S_j ∈ CT.prioridade faça
8:    Conjunto_Pedido ← Conjunto_Pedido ∪ (S_j, T)
9: fim para
10: para todo S_j ∉ Processos_Locais e S_j ∉ CT.prioridade faça
11:    Conjunto_Pedido ← Conjunto_Pedido ∪ (S_j, T)
12: fim para
13: para todo S_j ∉ Processos_Locais e S_j ∈ CT.prioridade faça
14:    Conjunto_Pedido ← Conjunto_Pedido ∪ (S_j, T)
15: fim para
16: Atualiza_Prioridades ⟨Conjunto_Pedido⟩
17: para todo tupla (S_j, T) ∈ Conjunto_Pedido faça
18:    Envia ⟨Requisicao_Token, T⟩ para S_j
19:    para todo token t_j ∈ T faça
20:       CT[t_j] ← self
21:    fim para
22: fim para
23: Aguarda pelo recebimento dos tokens
24: {...}
25:
```

Observe na Figura 3.9, que o processo P_i selecionou a instância T_5 do recurso R_2 que estava com o processo local P_k. O critério considerado aqui, foi devido à localização do processo P_k, que resulta em um menor custo de comunicação em relação à outra opção, a instância mantida pelo processo remoto L_b.

Ainda no procedimento de seleção de recursos, na Figura 3.10 é adicionada a lista C ao *Control Token*, que é responsável por associar prioridades aos processos que usam freqüentemente certos recursos. No exemplo em questão, P_i precisa selecionar dois dos três *tokens* que estão na posse dos processos L_a, P_k e P_j. No entanto, como L_a possui prioridade sobre seu recurso, como pode ser visto na ilustração da lista C, as opções então passam a ser P_k e P_j.

A lista C mantém para cada recurso, a identificação do processo prioritário e um

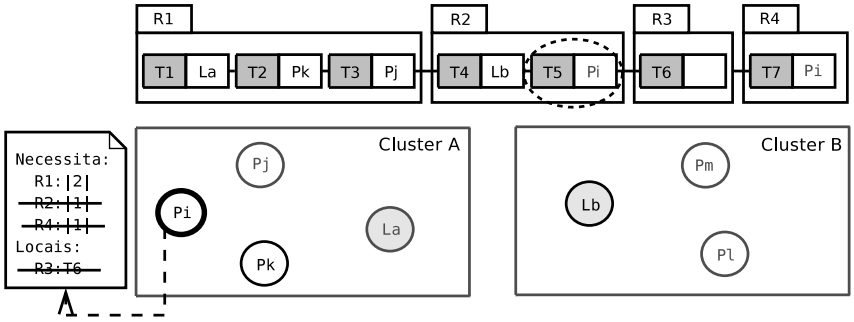

Figura 3.9: Seleciona *tokens* considerando localização

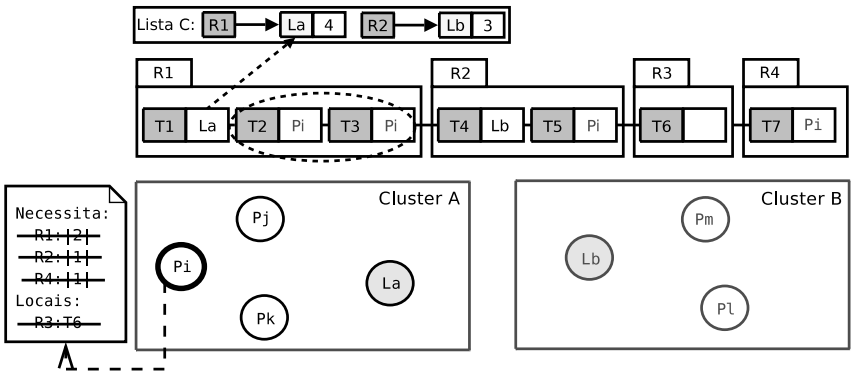

Figura 3.10: Seleciona *tokens* considerando prioridade

contador que corresponde ao número de solicitações sucessivas a um mesmo recurso que este processo realizou. Cada processo também mantém localmente um contador que é incrementado cada vez que ele necessita desse mesmo recurso. Assim, quando ele recebe o CT, ele verifica se possui prioridade sobre algum recurso e se insere no CT como prioritário. Caso o processo verifique que não precisará mais da prioridade sobre o recurso, ele retira sua identificação da lista de prioridades no CT e reinicia seu contador local. Um processo é considerado um candidato a ter prioridade sobre um recurso quando tal contador alcança um valor pré-definido, que é um parâmetro do algoritmo e que pode ser configurado à critério da aplicação.

Ao obter a posse do CT, um processo candidato pode atualizar a lista C em um dos seguintes casos: quando seu contador local supera o limite mínimo pré-estabelecido e não há nenhum processo com prioridade sobre o recurso, quando seu contador local é maior que o valor correspondente encontrado no CT, ou então, quando seu contador local é inferior ao valor pré-definido e ele deixa de ter prioridade sobre o recurso. O Algoritmo 9 apresenta os detalhes deste procedimento.

Após selecionar os recursos necessários, o processo P_i envia os pedidos para os processos, os quais estão com os *tokens* que ele selecionou. Veja na Figura 3.11 o exemplo

Algoritmo 9 - Atualiza as prioridades dos processos (Processo S_i)

```
1: Atualiza_Prioridades(Conj_Pedido):
2: para todo token t tal que 0 ≤ t < k faça
3:    se prioridade_local[t] > 0 e t ∈ tokens_necessarios então
4:       prioridade_local[t] ← prioridade_local + 1
5:    senão
6:       prioridade_local[t] ← 0
7:       CT[t].prioridade[S_i] ← 0
8:    fim se
9:    se prioridade_local[t] ≥ LIMITE_MINIMO_PRIORIDADE então
10:      CT[t].prioridade[S_i] ← prioridade_local[t]
11:   fim se
12:   para todo processo S ∈ Conj_Pedido faça
13:      CT[t].prioridade[S] ← 0
14:   fim para
15: fim para
16:
```

desse passo.

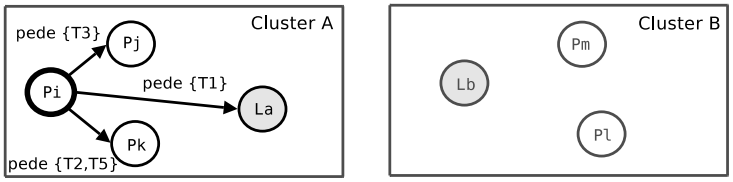

Figura 3.11: Envia pedidos de recursos aos processos selecionados

O Algoritmo 10, descreve o recebimento de uma mensagem de pedido de *tokens*. Ao receber essa mensagem, o processo envia imediatamente todos *tokens* requisitados, dos quais estão livres, em uma mensagem chamada *ACK1*. Essa mensagem é enviada mesmo estando vazia, ou seja, se não houver nenhum *token* livre para enviar. Se alguns dos *tokens* requisitados estão travados ou ainda não estão disponíveis localmente, uma segunda mensagem chamada *ACK2* é enviada, contendo o restante dos *tokens* requisitados, quando estes estiverem disponíveis.

Observe no Algoritmo 11, que quando um processo recebe todas as mensagens de *ACK1*, ele então trava os *tokens* recebidos e passa a estar apto a enviar o CT ao próximo processo da fila (linhas 5 e 6). Além disso, caso ele libere o CT a um processo remoto ele também deve atribuir o valor 0 à variável *num_preempt*, que é o contador de preempções realizadas. Se o processo receber nas mensagens *ACK1* todos os *tokens* que ele requisitou, ele entra em sua seção crítica (linhas 7 e 8). Caso contrário, ele aguarda o recebimento das mensagens *ACK2* que contêm o restante dos *tokens* solicitados e em seguida, ele pode entrar em sua seção crítica (linhas 10 a 12). Veja também um exemplo desses procedimentos na Figura 3.12.

Algoritmo 10 - Recebe pedido de token

1: **Recebe_Requisicao_Token(T, S_j):**
2: **para todo** token $t \in T$ **faça**
3: **se** $travado(t) = FALSO$ e $t \in tokens\_locais$ **então**
4: $ACK1 \leftarrow ACK1 \cup t$
5: $tokens\_locais \leftarrow tokens\_locais - t$
6: **senão**
7: $Q \leftarrow Q \cup t$
8: **fim se**
9: **fim para**
10: **se** $Q \neq \emptyset$ **então**
11: $tokens\_devidos \leftarrow tokens\_devidos \cup Q$
12: **fim se**
13: Envia $\langle Mensagem\_ACK, ACK1 \rangle$ para S_j
14:

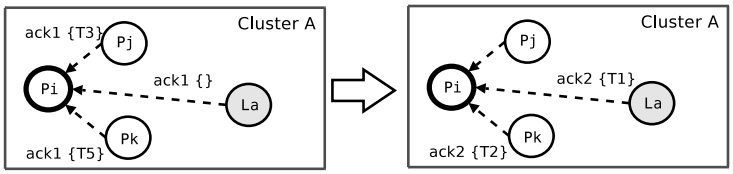

Figura 3.12: Recebe as mensagens de ACK1 e ACK2

Algoritmo 11 - Recebe mensagem de ACK (ACK1 e ACK2)

1: **Recebe_Mensagem_ACK(ACK, S_j):**
2: **para todo** token $t \in ACK$ **faça**
3: $tokens\_locais \leftarrow tokens\_locais \cup t$
4: **fim para**
5: trava $\langle tokens\_locais \rangle$
6: {Libera o envio do *Control Token* em caso de pedidos ...}
7: **se** $tokens\_necessarios \subseteq tokens\_locais$ **então**
8: {Acessa a seção crítica}
9: **senão**
10: {Aguarda o ACK2 ...}
11: trava $\langle tokens\_locais \rangle$
12: {Acessa a seção crítica}
13: **fim se**
14:

Ao liberar a seção crítica o processo executa o procedimento $Libera\_SC()$, que pode ser visto no Algoritmo 12. Caso ainda possua o CT, o processo então pode enviá-lo ao próximo processo da fila distribuída definida pela variável *next*, se ela não estiver vazia. Além disso, se o processo para o qual o CT está sendo enviado for um processo remoto, ele atualiza a variável *owner* com a identificação desse processo remoto. Por fim, ele envia as mensagens de *ACK2* para todos os processos que estão esperando pelos *tokens* e marca localmente que os *tokens* não estão mais presente.

Algoritmo 12 - Libera a seção crítica

```
1: Libera_SC():
2: requesting ← FALSO
3: se control_token = VERDADE e next ≠ 0 então
4:    se next ∉ Processo_Locais então
5:       num_preempt ← 0
6:       se owner = 0 então
7:          owner ← remote_owner
8:       fim se
9:       remote_owner ← 0
10:   fim se
11:   Envia ⟨ControlToken, CT, num_preempt⟩ para next
12:   next ← ∅
13:   control_token ← FALSO
14: fim se
15: para todo tupla(S_j, ACK2) ∈ tokens_devidos faça
16:   tokens_locais ← tokens_locais − ACK2
17:   Envia ⟨Mensagem_ACK, ACK2⟩ para S_j
18: fim para
19: tokens_devidos ← ∅
20:
```

3.3 Análise de Complexidade

A análise de complexidade apresentada nesta seção é baseada no número total de mensagens trocadas para que um processo obtenha o direito de executar sua seção crítica.

A complexidade de mensagens obtida pelo algoritmo proposto foi baseada na análise feita pelos trabalhos de Bertier *et al.* [8], Boubadallah-Laforest [9] e Naimi-Trehel [38], os quais se relacionam com o trabalho proposto devido às técnicas empregadas, inspiradas nestes trabalhos.

Para que um processo obtenha os recursos necessários para acessar sua seção crítica, o número total de mensagens que ele precisa trocar, no algoritmo proposto, varia de 0 a $n + 3k$, onde k é o número de recursos compartilhados e n é o número total de processos do sistema distribuído. Porém, na média, para um valor k constante, a complexidade é $O(log(n))$.

Para uma análise mais detalhada dessa complexidade, considere os seguintes casos.

- Se um processo P_i encontra todos os *tokens* que ele necessita localmente, ou seja, no seu conjunto $tokens\_locais_i$, então ele entra em sua seção crítica sem ter enviado nenhuma mensagem.

- Se um processo P_i não possui todos os *tokens* localmente. Então, no algoritmo proposto, o processo precisa solicitar o *Control Token* usando o mesmo mecanismo

de busca de *token* utilizado por Bertier *et al.* [8], que por sua vez foi baseado no algoritmo de Naimi e Trehel [38]. Como esse processo obtem o *Control Token* em um espaço finito de tempo, considere as seguintes possibilidades:

a. Assim como nos trabalhos citados [38, 8], o número de mensagens usadas para obter o *Control Token* (ou *token* no caso das referências) varia de 0 a $n-1$ e é $O(log(n))$ na média, devido às características da estrutura lógica em árvore discutidas anteriormente.

b. Quando o processo P_i recebe o *Control Token*, ele entra em sua seção crítica se encontrar todos os *tokens* necessários no conjunto de *tokens* livres do *Control Token*. Por esta razão, o número de mensagens utilizadas no total é o mesmo do item **a**.

c. Caso um subconjunto de *tokens* necessários não esteja disponível no *Control Token*, então o processo P_i enviará mensagens solicitando esses *tokens* aos processos que os possuem ou que os possuirão. Assim, é possível ainda ocorrer os casos à seguir:

[1.] O processo P_i precisará enviar no máximo k mensagens pedindo os *tokens*, que equivalem às instâncias de recursos, para os processos que os mantêm, e conseqüentemente receber no máximo k mensagens de $ACK1$ como resposta, contendo todos os *tokens* que ele necessita ou não. Assim, são adicionadas k mensagens ao resultado do item **a**.

[2.] Caso o processo P_i não receba todos os *tokens* necessários nas mensagens de $ACK1$, ele então irá recebê-los nas mensagens de $ACK2$, que também poderão ser no máximo k. Assim, além das mensagens utilizadas para obter o *Control Token* (item **a**), um processo usa no máximo $3k$ para obter o direito de executar sua seção crítica.

Capítulo 4

Relógios Lógicos

O algoritmo distribuído proposto neste trabalho é baseado em um modelo assíncrono em computação distribuída. O modelo assíncrono se caracteriza pelos fatos de que cada processo possui uma base de tempo própria e independente das dos outros processos, chamada de relógio local, e de que o atraso que uma mensagem sofre para ser entregue entre dois processos é finito, mas imprevisível. Este modelo é considerado mais realista, pois reflete algumas das características dos sistemas distribuídos atuais.

Para avaliar a qualidade de um algoritmo é útil medir o seu consumo de recursos. Quanto menor o consumo, melhor o algoritmo. Dependendo do tipo de computação, tais recursos podem incluir a quantidade de memória, ciclos de processador, número de processadores, número de mensagens enviadas e vários outros recursos sobre os quais a demanda é mais relevante. No caso dos algoritmos distribuídos, a avaliação de complexidade considera, geralmente, as mensagens enviadas e o tempo. As medidas são expressas na forma de "pior-caso", que é a maior complexidade de qualquer computação no algoritmo.

A complexidade de mensagens é dada pelo número de mensagens enviadas entre processos durante a computação no pior caso considerando-se variações possíveis na estrutura de conexão dos processos e nas execuções do algoritmo, que poderiam produzir diferentes conjuntos de eventos.

Em um modelo assíncrono, a complexidade de tempo deve somente considerar o tempo relacionado à comunicação. Assume-se que a computação local não consome tempo e que o tempo para envio de uma mensagem é de uma unidade de tempo. A complexidade de tempo então seria o número de mensagens enviadas na maior cadeia causal da forma "recebo uma mensagem e envio uma mensagem como conseqüência" que ocorre em todas as execuções do algoritmo e sobre todas as variações na estrutura de conexão dos processos.

A complexidade de tempo somente considera as mensagens que ocorrem "seqüencialmente" relativas a eventos de recebimento e envio que possuem uma dependência causal. Assim, não são consideradas as mensagens relacionadas a eventos concorrentes.

Uma abstração útil consiste em modelar a computação que acontece localmente em um processo como uma seqüência de eventos. Associados com um evento estão: a identificação do processo onde ele ocorre, o tempo local no qual o evento acontece, assim as possíveis mudanças no estado local do processo decorrentes ou do envio (recebimento) de mensagens para (de) vizinhos ou da execução de uma operação que não envolva interação com outro processo, chamado de evento interno. Em um evento só se pode receber no máximo uma mensagem, embora nenhuma restrição exista em relação ao número de mensagens enviadas associadas a este evento. A computação distribuída que acontece é então naturalmente associada com o conjunto de eventos que ocorrem em todos os processos, e que denotamos por V.

Apesar de o sistema ser totalmente assíncrono, existe uma relação de interdependência entre os diversos eventos que constituem a computação, definida por Lamport [30].

Considere, por exemplo, que os eventos v_1, $v_2 \in V$. Dizemos que v_1 *"aconteceu-antes"* de v_2, se e somente se os eventos v_1 e v_2 ocorrem no mesmo processo, p_i por exemplo, nos tempos locais $t_i^1 < t_i^2$ e nenhum outro evento ocorre naquele mesmo processo em um instante t_i' tal que $t_i^1 < t_i' < t_i^2$. Ou então, v_1 envolve o envio de uma mensagem m recebida através de v_2, sendo que v_1 e v_2 ocorrem em processos vizinhos.

Seja A uma relação chamada de *relação de causalidade* ou *"aconteceu-antes"* [30, 6], onde $v \in V$. Se os eventos $v_1 A v_2$ e $v_2 A v_3$ então $v_1 A v_3$. A relação A é transitiva e irreflexiva, formando, portanto, uma ordem parcial sobre os eventos de V.

Podemos descrever as ocorrências de eventos sobre o tempo através de um diagrama de tempo. Assim, processos são representados por eixos de tempos locais, nos quais os tempos locais crescem da esquerda para a direita e setas representam mensagens. Esta representação é utilizada no decorrer deste capítulo. Na Figura 4.1, são representados dois processos, i e j, e podemos observar que $v_1 A v_2, v_2 A v_3$ e $v_1 A v_3$, segundo a definição da relação de causalidade.

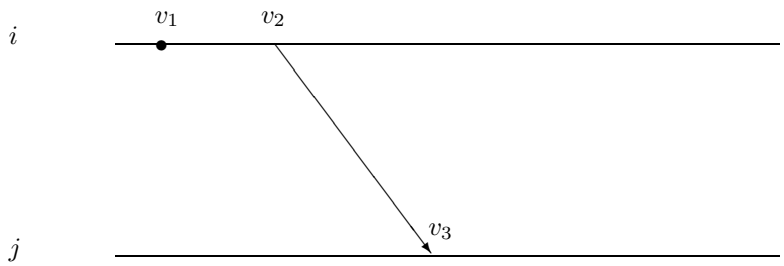

Figura 4.1: Relação *aconteceu-antes*

4.1 Relógios Lógicos

Para a compreensão adequada da execução de um programa distribuído é importante determinar a ordem causal entre os eventos que ocorrem na computação. A concorrência entre eventos e o não-determinismo em programas distribuídos são questões importantes na análise, monitoração, depuração e visualização destes eventos.

Em computações distribuídas, os relógios devem ser definidos de forma a expressar a causalidade entre os eventos. A caracterização e representação de relação de causalidade entre os eventos não é um problema trivial. Consideramos que um relógio é uma função T de eventos para um conjunto ordenado $(V, <)$, tal que $vAv' \Rightarrow T(v) < T(v')$.

Caso considerássemos $T(v)$ o instante de tempo real no qual o evento v ocorre, não conseguiríamos caracterizar a causalidade entre os eventos, porque $T(v) < T(v')$ não necessariamente implicaria em vAv'. Um problema adicional é que, geralmente, não estão disponíveis em sistemas distribuídos um conjunto de relógios locais de tempo real sincronizados.

Lamport [30] apresentou um relógio que atribui a um evento v o comprimento k da maior seqüência $(e_1, e_2, \cdots, e_k)$ de eventos que satisfaz $e_1Ae_2A\cdots e_k = v$, tal que $T(e_1) < T(e_2) < \cdots < T(e_k)$.

O valor de T pode ser calculado para cada evento ocorrido em um processo p pelo algoritmo distribuído apresentado no Algoritmo 13.

No algoritmo apresentado não é especificado sob que condições uma mensagem deve ser enviada ou como o estado de um processo muda. O relógio é um mecanismo suplementar que pode ser acrescentado a qualquer algoritmo distribuído para ordenar seus eventos. A cada execução de um evento o relógio local do processo é incrementado. A cada mensagem enviada é anexado o relógio local do processo. Além disso, ao receber uma mensagem, o processo atualiza o seu relógio local com o valor máximo entre o valor

Algoritmo 13 - Ações em p para o algoritmo `RELÓGIO DE LAMPORT`:

(1) Para um estado inicial:
$T_p \leftarrow 0$
(2) Regra para envio de mensagem:
$T_p \leftarrow T_p + 1$
Envie $\langle msg, T_p \rangle$;
Mude o estado
(3) Regra para recebimento de mensagem de um processo $i \in N_p$:
$T_p \leftarrow max(T_p, T_i) + 1$
Mude o estado
(4) Regra para evento interno:
$T_p \leftarrow T_p + 1$
Mude o estado

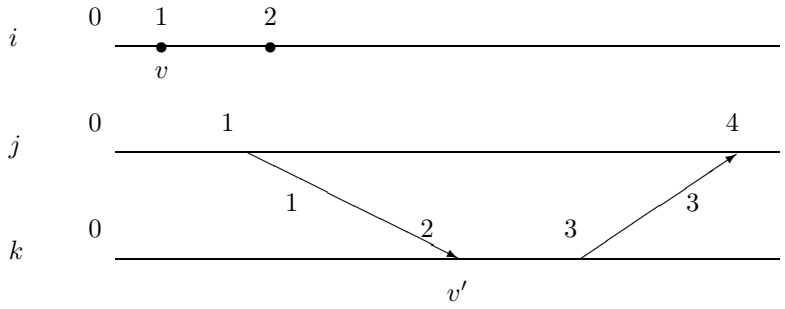

Figura 4.2: Relógio de Lamport

de relógio recebido e o seu próprio e incrementa de um.

A Figura 4.2 ilustra os valores dos relógios locais dos processos i, j e k no decorrer de suas execuções.

Em alguns casos, é útil que os relógios expressem não somente a ordem causal entre os eventos, mas também a concorrência entre eles. Uma importante consideração a respeito do relógio de Lamport, é que ele satisfaz a propriedade $v \rightarrow v' \Rightarrow T(v) < T(v')$, no entanto o oposto não é verdadeiro. Observe na Figura 4.2 que $T_i(v) = 1 < T_k(v') = 2$, entretanto não se pode dizer que vAv'. Fidge [20] e Mattern[35] propuseram um mecanismo conhecido como relógio de vetor, que satisfaz a seguinte propriedade: $v \rightarrow v' \Leftrightarrow T(v) < T(v')$. Neste caso, um vetor é associado a cada estado local de cada processo tal que a ordem parcial dos eventos é preservada.

Considere a existência de n processos identificados por inteiros com valores entre 1 e n. O relógio de vetor T_p é atualizado em um processo $p \in 1, \cdots, n$ segundo as regras descritas no Algoritmo 14.

No Algoritmo 14 um processo incrementa seu próprio componente do vetor depois de cada evento. Além disso, ele inclui seu relógio vetor em cada mensagem enviada. Na recepção de uma mensagem, ele atualiza seu vetor com o máximo entre os componentes

Algoritmo 14 - Ações em p para o algoritmo RELÓGIO DE VETOR:

(1) Para um estado inicial:
$(\forall i : (i \neq p) : T_p[i] = 0) \wedge (T_p[p] = 1)$
(2) Regra para envio de mensagem:
$T_p[p] \leftarrow T_p[p] + 1$
Envie $\langle msg, T_p \rangle$;
Mude o estado
(3) Regra para recebimento de mensagem de um processo $i \in N_p$:
Receba $\langle msg, T_i \rangle$;
para $j = 1$ **até** n **faça**
  $T_p[j] \leftarrow max(T_p[j], T_i[j])$
fim para
$T_p[p] \leftarrow T_p[p] + 1$
Mude o estado
(4) Regra para evento interno:
$T_p[p] \leftarrow T_p[p] + 1$
Mude o estado

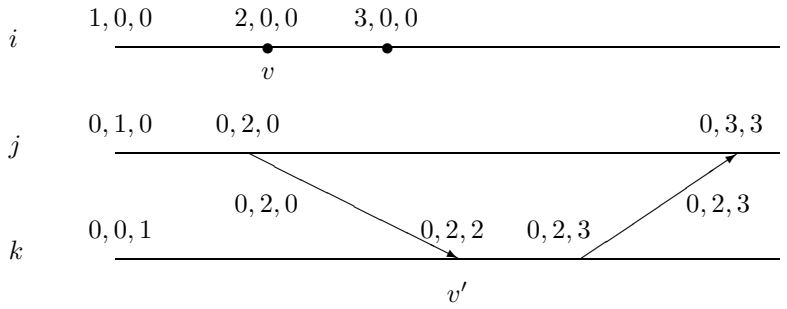

Figura 4.3: Relógio de Vetor

dos vetores e incrementa o elemento correspondente a sua própria posição.

A Figura 4.3 ilustra a execução de tal algoritmo. Repare que não ocorre $T_k(v) =$ (2,0,0) $< T_i(v' =$ (0,2,2) nem $T_i(v') =$ (0,2,2) $< T_k(v) =$ (2,0,0) e que v e v' são eventos concorrentes.

Os relógios de vetores têm sido tipicamente empregados em algoritmos que auxiliam a depuração e análise de programas paralelos distribuídos. Existem ainda generalizações da noção de relógios vetores, chamados relógios matrizes [18], que permitem a representação da precedência causal com um nível maior de profundidade cuja aplicação notória pode ser encontrada na área de bancos de dados.

4.2 Relógios Lógicos como Estratégia de Medição de Desempenho

Em ambientes de *Grids*, o tempo de relógio de parede de duas execuções de um mesmo programa pode variar dependendo da carga externa nas máquinas. Medir o desempenho

em tais ambientes é uma tarefa complexa e não garante a total confiabilidade dos resultados devido a tais variações de tempo. Por esta razão, utilizamos o modelo de relógio lógico como uma estratégia adicional para medir o desempenho dos algoritmos.

Assim como sugerido em Drummond e Barbosa [18], foi adaptado neste trabalho um modelo de relógio lógico a fim de medir o tempo de espera para um processo entrar em sua seção crítica. O emprego de relógios lógicos é baseado em relógio de vetores, como proposto em [20, 35] com algumas mudanças para refletir a diferença de latência dos canais de comunicação em um ambiente de *Grid*.

Consideramos aqui a maior cadeia com relação de causalidade para que um processo obtenha a mensagem que concede a ele o direito de executar sua seção crítica. A unidade de tempo contabilizada no relógio lógico proposto é dada pela latência hipotética dos canais de comunicação que interligam os processos remotos. A latência nos canais que interconectam processos que estão em um mesmo *cluster* é desconsiderada, para que seja possível medir a influência das latências nos canais remotos.

Um exemplo desta idéia é ilustrado na Figura 4.4. Observe a cadeia causal formada, no "pior-caso", envolvendo os processos que solicitaram o acesso à seção crítica, e o retorno da mensagem onde é contabilizado o tempo lógico esperado por cada processo.

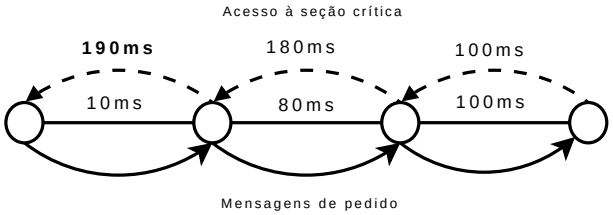

Figura 4.4: Exemplo de Execução do Relógio Lógico

O mecanismo de relógios lógicos adaptado por esse trabalho foi utilizado estritamente para medição de desempenho dos algoritmos testados, e não na sincronização ou ordenação de mensagens como utilizado em [30]. Ou seja, o uso de Relógios Lógicos não altera a execução dos algoritmos, apesar de serem utilizadas estruturas acopladas às mensagens para sua monitoração.

No algoritmo proposto, para executar uma seção crítica, um processo, a partir do momento que requisita acesso à sua seção crítica precisa esperar pelo *Control Token* e pelos *tokens* dos recursos. O tempo total requerido para um processo entrar em sua seção crítica é obtido pela soma dos tempos de espera. No algoritmo proposto, mensagens de requisição ao *Control Token* de um processo seguem o mesmo caminho definido pelas mensagens de

Control Token, possivelmente composto de conexões com diferentes latências. Então, para monitorar a espera ao *Control Token*, ele precisa somente monitorar as mensagens de *Control Token*. Analogamente, uma requisição a um *token* de recurso pode chegar a um processo que ainda não o obteve. Por exemplo, um processo que também espera por mensagens *ACK2* para entrar em sua seção crítica e então ao sair ele envia as mensagens *ACK2* correspondente, formando uma corrente de mensagens *ACK2*.

Considerando estes fatos, propusemos um relógio lógico que tem como objetivo a captura do maior caminho encadeado de mensagens de *tokens* com relação causal. Ele foi construído fazendo com que cada processo P_i mantivesse n elementos no vetor, V_i, coletados ao longo das mensagens de *tokens*. Como dois tipos de mensagens são monitorados, *Control Token* e *ACKs*, dois vetores são necessários, no entanto as regras de atualização deles são similares.

As regras para atualizar o vetor do *Control Token* são descritas no Algoritmo 15.

Algoritmo 15 - Adaptação do Relógio Lógico - Processo P_i

1: **(R.1) Ao receber uma mensagem de Control Token de um processo P_j (Control Token, V_j):**
2: $controltoken\_waiting\_time \leftarrow V_j[i]$;
3: **para todo** $k \in V_i$ **faça**
4: $V_i[k] \leftarrow \max(V_i[k], V_j[k])$;
5: **fim para**
6: $V_i[i] \leftarrow 0$;
1:
2: **(R.2) Ao enviar uma mensagem de Control Token a um processo P_j em uma conexão com latência x:**
3: **para todo** $k \in V_i$ **faça**
4: $V_i[k] \leftarrow (V_i[k] + x)$;
5: **fim para**
6: Envia $\langle ControlToken, V_i \rangle$ para P_j

Embora os algoritmos para atualização do *Control Token* e dos vetores *ACKs* sejam os mesmos, os métodos para o cálculo do tempo de espera para cada *token* de recurso e para o *Control Token* não são iguais. No caso de mensagens dos *tokens* de recursos, deve-se considerar o maior tempo de espera acumulado das mensagens *ACK1* e *ACK2*. Assim, no Algoritmo 15, a linha 2 de R.1 é substituída por:

"$resourcetoken\_waiting\_time \leftarrow \max(resourcetoken\_waiting\_time, V_j[i])$"

O *resourcetoken_waiting_time* precisa também ser comparado com o maior tempo de espera acumulado de todas mensagens *ACK1*.

Como a cadeia de espera para o *Control Token* e os *tokens* de recursos ocorrem seqüencialmente, no pior caso o tempo necessário para que um processo P_i execute

sua seção crítica, é a soma do tempo em *controltoken_waiting_time* com o tempo em *resourcetoken_waiting_time*.

Veja na Figura 4.5 um exemplo envolvendo quatro mensagens trocadas por quatro processos conectados através de *links* com diferentes latências, e os relógios lógicos correspondentes. Nessa figura, quatro processos P_k, P_a, P_i e P_j executam o envio de uma mensagem entre eles. Repare na figura, que as setas representam uma mensagem com dependência causal entre os processos. Note que a cada envio da mensagem de um processo para outro, o vetor lógico é incrementado pela latência do canal que interliga esses processos. Na figura, é possível também observar para cada processo, uma imagem de seu vetor atualizado localmente, onde os valores em negrito representam os tempos lógicos utilizados para que a mensagem chegasse até ao processo.

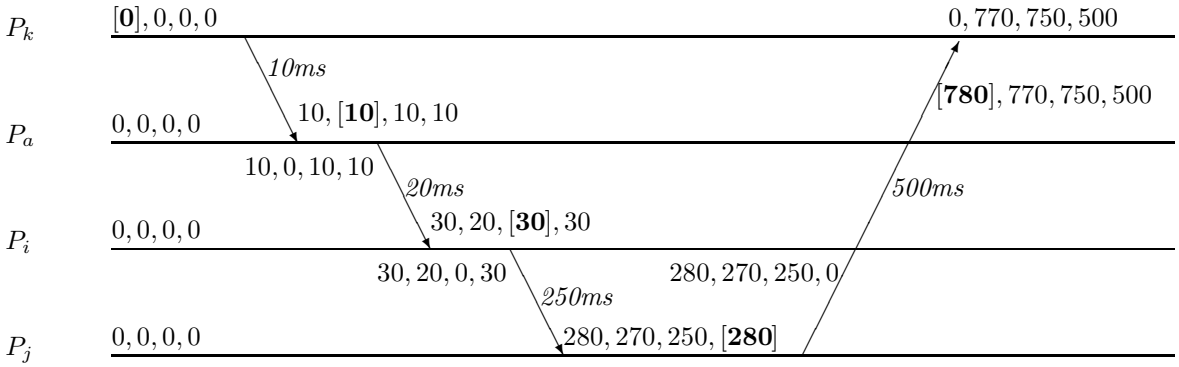

Figura 4.5: Relógio de Vetor Proposto

Nos experimentos usando o mecanismo de relógios lógicos, consideramos que as mesmas instâncias de recursos são requisitadas em cada seção crítica, porém diferentes para cada processo.

Capítulo 5

Resultados Experimentais

Este capítulo apresenta uma análise do desempenho dos algoritmos propostos em relação aos existentes na literatura. Com o intuito de atestar o bom desempenho dos algoritmos desenvolvidos neste trabalho, foram utilizados diversos cenários com diferentes parâmetros e configurações físicas, comparando os algoritmos propostos com o algoritmo de Bouabdallah-Laforest [9] e uma generalização.

5.1 Ambiente de Testes

Os algoritmos apresentados neste trabalho foram implementados na linguagem Python versão 2.5, usando a ferramenta PyMPI como uma camada de integração entre o interpretador Python e a biblioteca de troca de mensagens MPI, executando sobre a implementação LAMMPI (versão 7.1.4) usada nos testes. A escolha da linguagem Python para o desenvolvimento do algoritmo foi feita devido à necessidade de se ter uma linguagem de desenvolvimento rápido, com uso de diversos componentes de alto nível, como estrutura de dados e funções simplificadas para manipulação de arquivos, além de possuir uma abstração útil na integração com a biblioteca MPI.

Os experimentos apresentados neste trabalho foram executados em dois ambientes computacionais distintos, que foram:

- **Ambiente de testes 1:** Neste ambiente, foram utilizadas 24 estações de trabalho bi-processadas, com processadores Intel Xeon 3.06 GHz com 512 KB de cache e AMD Opteron 248 2.2 GHz e 1024KB de cache. Foram utilizados 60 processadores no total, contando com os núcleos dos processadores *dual core*. Todas as estações

possuem instalado o sistema operacional GNU/Linux com kernel 2.4.21 compilado com suporte a SMP *(Symmetric Multi Processor)*, o que permite que o sistema operacional gerencie os processadores para que eles sejam utilizados em paralelo. As estações estão conectadas a uma rede Ethernet de 1.0 Gbits/s com latência próxima a zero.

- **Ambiente de testes 2:** Neste segundo ambiente computacional, utilizamos 24 computadores, cada um com um processador Intel Pentium IV com 2.6 GHz de freqüência, 512 MB de memória RAM, sistema operacional GNU/Linux com kernel 2.6.8, conectados por uma rede ethernet de 1.0 Gbits/s.

Nessa dissertação de mestrado são apresentados dois métodos que foram usados para medição de desempenho dos algoritmos distribuídos. O primeiro método consiste em medições utilizando tempo de relógio de parede ou relógio físico. Este método é usado para medir o tempo que um processo demora para entrar em sua seção crítica (SC). O outro método utiliza o mecanismo de relógios lógicos, seguindo o conceito descrito no Capítulo 4.

Inicialmente, para emular um ambiente *Grid* usando o **Ambiente de testes 1**, foi criada uma topologia lógica que consiste de 48 processadores divididos entre três *clusters* (C0, C1 e C2), contendo em cada *cluster* 16 processadores. Esta topologia lógica foi criada com base no número de processadores físicos disponíveis, optando assim por alocar cada processador lógico a um processador físico. Para o **Ambiente de testes 2**, que é composto de 24 processadores físicos, utilizamos a topologia lógica de 24 processadores divididos entre três *clusters*, contendo 8 processadores em cada *cluster*. Dessa forma, foi mantida a estratégia de alocar cada processador lógico a um processador físico.

Em alguns dos testes realizados, variamos as latências dos canais de comunicação entre os processadores, simulando processadores distantes geograficamente ou que se comunicam por canais mais lentos. Para simular estas latências, foram utilizadas duas estratégias. A primeira utiliza a função de *sleep* da linguagem Python, que atrasa a execução de um código de acordo com a quantidade de segundos informados no parâmetro da função. A segunda, utiliza a ferramenta NetEm [1] e [25]. O NetEm é uma ferramenta que permite emular propriedades de redes. Ele trabalha como um módulo do kernel, permitindo inserir atrasos diretamente nos pacotes de entrada e saída da rede. A ferramenta NetEm atualmente é disponibilizada em conjunto com o pacote de ferramentas para redes e controle de tráfego *iproute2*[2].

Os resultados desta avaliação são uma média de 10 execuções. Em cada execução um processo entra em sua seção crítica por 10 vezes. Em cada vez, ele solicita um conjunto de recursos, ao recebê-lo, ele entra em sua seção crítica permanecendo por um tempo $\boldsymbol{\alpha}$ e após sair dela, libera os recursos. Em seguida, esse processo aguarda ainda um tempo $\boldsymbol{\beta}$ para fazer uma nova solicitação de recursos para entrar novamente em sua seção crítica. Para simular os pedidos de recursos feitos pelos processos, foi criado um arquivo com todos os recursos e instâncias gerados aleatoriamente. Neste arquivo é definido um conjunto de recursos e instâncias que cada processo solicita para entrar em sua seção crítica. O mesmo arquivo é usado nos testes dos dois algoritmos.

Por fim, no algoritmo proposto, quando um processo remoto solicita o *Control Token*, é criada uma fila de prioridade que faz com que os processos locais posterguem o envio do *Control Token* ao nó remoto pelo número de vezes definido no parâmetro "Limite de Preempções". Essa estratégia permite que processos locais recebam prioridade em relação a processos remotos, reduzindo com isso, o envio de mensagens remotas pelos canais de comunicação. Porém, atingido o valor limite, as requisições remotas que estão na fila são imediatamente atendidas, evitando assim o problema de *starvation*. O valor definido neste parâmetro equivale a 50% do número de processos em cada *clusters*.

Nos testes realizados utilizando relógios lógicos, não se contabiliza no tempo lógico os tempos de α e β. Apesar de eles influenciarem no resultado final do algoritmo, estes só tem valor significativo nos tempos contabilizados por relógios físicos.

Inicialmente, comparamos a primeira versão do algoritmo proposto com o algoritmo de Bouabdallah e Laforest, que resolve o problema de exclusão mútua com múltiplos recursos compartilhados, porém com uma única instância por cada recurso. Em seguida foi feita uma adaptação do algoritmo de Bouabdallah e Laforest para atender o caso generalizado deste mesmo problema, utilizando, porém, um ambiente com múltiplos recursos e múltiplas instâncias para cada recurso, assim como proposto em seu artigo [9]. Esta adaptação foi comparada com a segunda versão do algoritmo proposto.

5.2 Algoritmo Proposto vs. Bouabdallah-Laforest para Múltiplos Recursos com Uma Instância

Nesta seção estão descritos os resultados dos testes realizados com a primeira configuração do algoritmo proposto neste trabalho. Os testes a seguir consideram o problema da exclusão mútua para o caso de haver múltiplos recursos e apenas uma instância por tipo

de recurso. Neste caso, um processo pode entrar em sua seção crítica a partir da obtenção de um subconjunto qualquer destes recursos. Processos podem entrar simultaneamente em suas respectivas seções críticas desde que mantenham o acesso exclusivo.

5.2.1 Variação da latência entre *clusters*

Este experimento possui o objetivo de investigar os efeitos da variação da latência entre *clusters*, simulando as mensagens que trafegam pelos canais de comunicação, que interconectam processadores geograficamente distantes, ou que sejam mais lentos.

A variação da latência é representada nas figuras, no eixo das abscissas, variando-se em 1ms, 250ms, 500ms e 750ms. No eixo das ordenadas, é mostrada a média dos tempos representados em milisegundos. Para as latências de comunicação local, consideramos um tempo constante para efeito de testes de 1ms.

Para avaliar os algoritmos testados, utilizamos os dois mecanismos de medição de desempenho citados anteriormente, e que são apresentados a seguir.

5.2.1.1 Comparação dos algoritmos usando Relógios Lógicos

Os resultados a seguir foram obtidos usando o **Ambiente de testes 1**.

O uso de relógios lógicos nos testes, como definido no Capítulo 4, permite que avaliemos o tamanho da maior cadeia de mensagens com relação de causalidade, no pior caso. Essa cadeia de mensagens representa o tempo lógico que um processo leva para obter acesso à sua seção crítica.

Nos testes de relógio lógico a seguir, cada processo sempre pede os mesmos recursos em todas as execuções.

A Figura 5.1(a) ilustra os valores dos tempos médios, obtidos através da soma das latências dos canais, até a obtenção do *Control Token*. Observe que as curvas do gráfico tendem a um crescimento, a medida que a latência entre *clusters* aumenta. No entanto, também podemos ver que na curva referente ao algoritmo proposto, o crescimento observado é muito menor, o que significa uma redução no tempo de obtenção do *Control Token* em comparação com o algoritmo de Bouabdallah-Laforest. Este resultado foi obtido através do mecanismo que prioriza requisições de processos locais em relação aos processos remotos, evitando o envio de mensagens remotas e criando assim, uma fila local por onde o *Control Token* possa ser passado. Veja também, na Figura 5.1(b), que as médias da

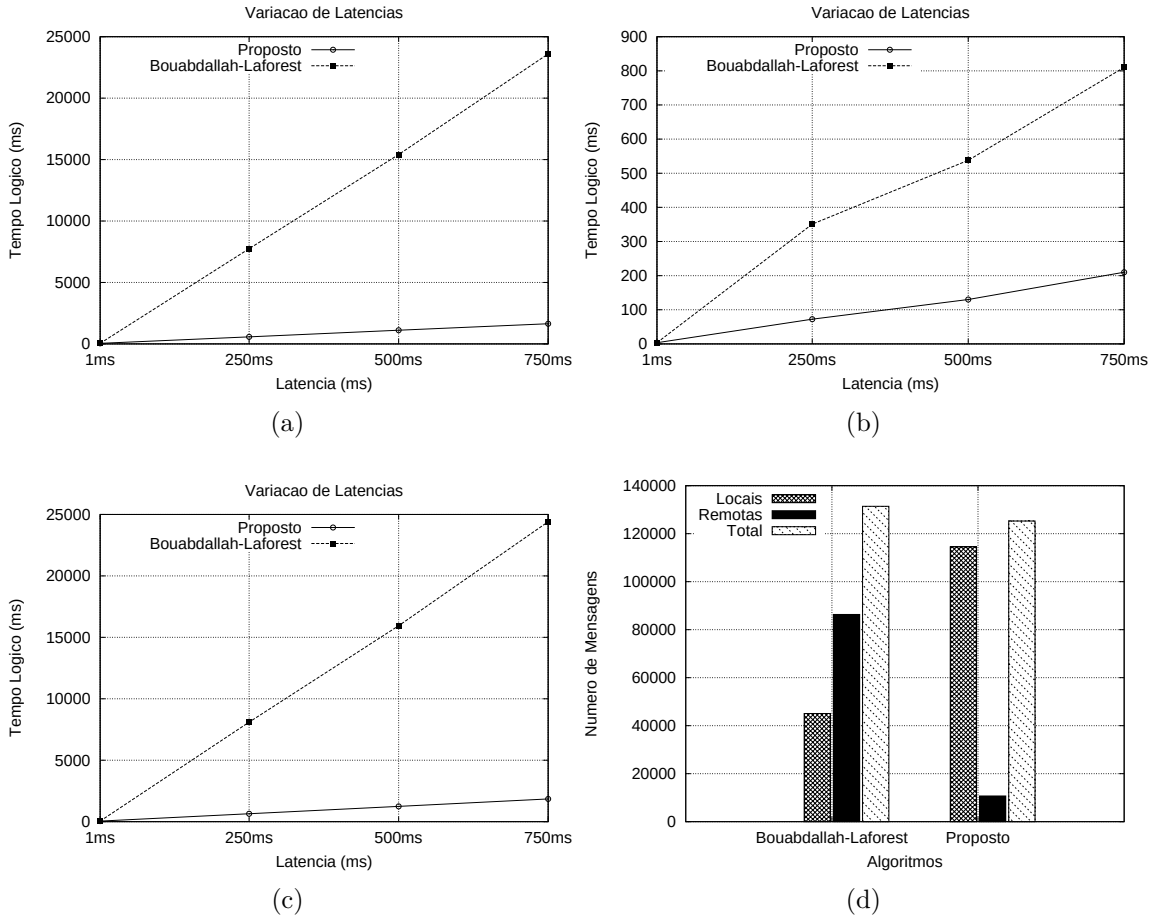

Figura 5.1: Variação da latência - Múltiplos Recursos com uma Instância: (a) Média do tempo lógico para obtenção do *Control Token* (b) Média do tempo lógico para obtenção do ultimo *token* (c) Média da soma dos tempos anteriores (d) Número de mensagens trocadas

soma das latências dos canais para a obtenção de cada *token*, no algoritmo proposto, também são reduzidas em relação ao algoritmo concorrente. Isto ocorre pelo procedimento anterior de obtenção do *Control Token* que faz com que processos locais sejam atendidos com maior prioridade, mantendo o *Control Token* por mais tempo localmente. Com isso, tem-se uma concentração maior dos *tokens* entre os processos locais, o que gera um menor custo de obtenção dos mesmos. O resultado apresentado na Figura 5.1(c), foi obtido a partir da soma das médias dos tempos lógicos para obtenção do *Control Token* e para obtenção dos *tokens* que representam os recursos.

O número de mensagens trocadas entre os processos é apresentado na Figura 5.1(d). Esta figura permite visualizar a relação entre mensagens remotas e mensagens locais trocadas pelos dois algoritmos em questão. Observe que o algoritmo proposto reduziu em cerca de 90% a quantidade de mensagens remotas em relação a mensagens locais, enquanto o algoritmo de Bouabdallah-Laforest enviou 47% mais mensagens por canais remotos do

que por canais locais, o que enfatiza os resultados obtidos nas figuras anteriores. Mesmo não sendo o objetivo da técnica utilizada, houve ainda uma pequena redução de 4.6% no número total de mensagens trocadas, utilizando o algoritmo proposto.

5.2.1.2 Comparação dos algoritmos usando Relógio Físico

Para os testes a seguir, foram utilizados os dois ambientes computacionais apresentados no início deste capítulo. Essa estratégia permite-nos observar o comportamento do algoritmo em ambientes de testes distintos, usando formas distintas para inserir as latências.

No **Ambiente de testes 1**, composto por 48 processadores heterogêneos, foram realizados testes inserindo as latências através da função *sleep* da linguagem Python. O atraso neste caso é inserido no ato do envio de uma mensagem para um canal logicamente configurado como distante. O objetivo aqui é medir o tempo que um processo leva para obter acesso a sua seção crítica a partir do momento que ele a solicita.

Note na Figura 5.2, que o aumento da latência resulta em um maior tempo para obter acesso à seção crítica, para ambos os algoritmos. Porém, tem-se um ganho de desempenho no algoritmo proposto, que consome um tempo muito menor de obtenção do que o algoritmo de Bouabdallah-Laforest. Veja que o crescimento das curvas em relação aos testes anteriores com relógio lógicos é semelhante, apesar de usarem medidas diferentes.

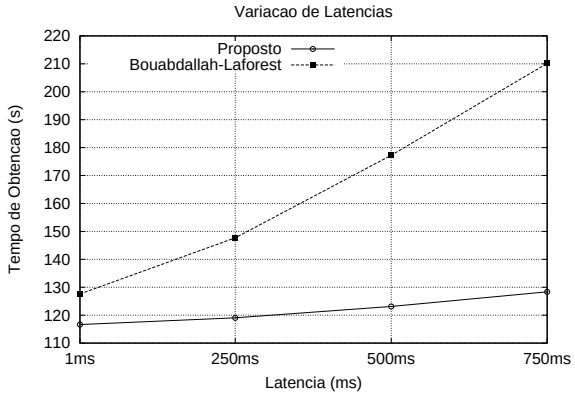

Figura 5.2: Tempo de Obtenção - usando *Sleep* - Mútiplos Recursos com uma Instância

Os valores apresentados na Figura 5.3, foram obtidos dos testes realizados no **Ambiente de testes 2**, que consiste em um ambiente com 24 processadores homogêneos. Assim como no teste anterior, foram inseridos atrasos nos canais de comunicação entre os processos para simulação da latência, no entanto, nesta avaliação foi utilizada a ferramenta NetEm, já mencionada no início deste capítulo. Como pode ser observado na

Figura 5.3, o algoritmo proposto também se mantém mais eficiente do que o algoritmo de Bouabdallah-Laforest, isso porque a curva que representa este último, apresentou ter uma tendência a um crescimento maior do que a curva obtida pelo algoritmo proposto. Isso mostra que, mesmo em ambientes computacionais diferentes, os resultados são bem relacionados.

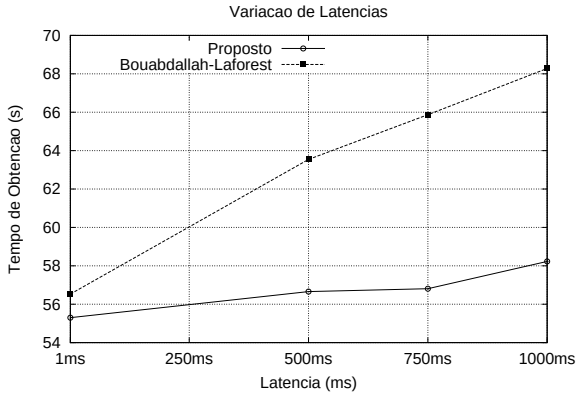

Figura 5.3: Tempo de Obtenção - usando NetEm - Múltiplos Recursos com uma Instância

Os resultados dos tempos médios de obtenção e desvio padrão usando relógio lógico e físico (com *sleep*), são apresentados na Tabela 5.1. Nessa tabela, é possível verificar que os valores percentuais do desvio padrão se mantiveram uniformes, não apresentando grande variação em relação aos valores crescentes nas médias dos tempos para obtenção da seção crítica. A Tabela 5.2, mostra o número de mensagens trocadas em cada variação realizada pelos algoritmos. Nessa tabela, é possível notar que o algoritmo proposto, em todas as variações, reduziu o número de mensagens remotas em relação às mensagens locais. Já o algoritmo de Bouabdallah-Laforest, enviou em número maior de mensagens por canais remotos do que por canais locais, o que por conseqüência, apresentou resultados inferiores. Pelo número de preempções mostrado nesta tabela, podemos observar que o mecanismo de preempções foi bem utilizado em todas as variações, este que permite que dado um limite de atendimentos à processos locais, processos remotos sejam atendidos, evitando assim, o problema de *starvation*.

5.3 Algoritmo Proposto vs. Bouabdallah-Laforest para Múltiplos Recursos com Múltiplas Instâncias

Nesta segunda parte dos testes, são apresentados os resultados obtidos com a segunda versão do algoritmo proposto e com uma generalização do algoritmo de Bouabdallah-Laforest, que consideram um ambiente com múltiplos recursos e múltiplas instâncias.

Tabela 5.1: *Tempos Médios e Desvio Padrão* - ***Variação de Latências - Múltiplos Recursos com Uma Instância***

Algoritmos	**Latências(ms)**	**Relógio Físico (s)**		**Relógio Lógico(ms)**	
		Média	**Desv. Pad (%)**	**Média**	**Desv. Pad (%)**
Algoritmo Proposto	1ms	116.66	4.77	47.59	0.58
	250ms	119.06	4.96	647.51	1.00
	500ms	123.12	4.99	1241.35	0.98
	750ms	128.37	4.96	1851.61	1.05
Bouabdallah Laforest	1ms	127.60	4.27	47.52	0.57
	250ms	147.70	4.86	8099.20	0.77
	500ms	177.26	5.85	15945.45	0.81
	750ms	210.14	5.55	24409.01	0.67

Tabela 5.2: *Tabela do Número de Mensagens Trocadas* - ***Variação de Latências - Múltiplos Recursos com Uma Instância***

Algoritmos	**Latências(ms)**	**Número de Mensagens**		**Preempções**
		Remotas	**Locais**	
Algoritmo Proposto	1ms	384.00	4107.14	88
	250ms	376.14	4092.14	85
	500ms	389.43	4090.00	88
	750ms	379.57	4074.43	86
Bouabdallah Laforest	1ms	3770.14	951.00	-
	250ms	2806.43	1917.29	-
	500ms	2842.00	1823.14	-
	750ms	2914.71	1744.71	-

Neste caso, um processo pode entrar em sua seção crítica com qualquer subconjunto de recursos e com quantas cópias desses que achar necessário, observando obviamente o acesso exclusivo destes.

Nessa bateria de testes, tentou-se realizar experimentos a fim de observar o comportamento dos algoritmos em diversas situações existentes em ambientes de *Grids*, explorando a variação de parâmetros dos algoritmos e configurações lógicas. Os principais experimentos foram: variação da latência entre *clusters*, variação do número de tipos de recursos disponíveis, variação do número total de processos.

5.3.1 Variação da latência entre *clusters*

Assim como nos testes apresentados na seção anterior, esta seqüência de testes tem o objetivo de verificar o comportamento e a eficiência dos algoritmos, a partir da variação da latência dos canais de comunicação que interligam *clusters* diferentes. Espera-se com este experimento, que o algoritmo proposto seja mais eficiente que o algoritmo concorrente, reduzindo o tráfego de mensagens através de canais remotos, a partir da concentração de atendimentos locais a processos que queiram acessar a seção crítica.

Os parâmetros utilizados nesta avaliação são os mesmos utilizados na versão anterior deste algoritmo. Lembrando neste caso, que os recursos podem ser de diferentes tipos, mas cada tipo pode possuir até 10 cópias de recursos, e ainda, um processo pode obter um subconjunto de quantos tipos e cópias de recursos ele necessitar. A distribuição desses recursos pelos processos é definida previamente em um arquivo de entrada.

Como nos testes anteriores, os valores das latências entre *clusters*, também, variaram em 1ms, 250ms, 500ms e 750ms. Para os valores de latências na comunicação local, consideramos um tempo constante para efeito de testes de 1ms. É importante salientar que, os tempos α e β não são somados no tempo lógico, apesar de influenciarem no resultado final destes.

As duas estratégias de medição de tempo, usando os dois tipos de relógios, lógico e físico, foram utilizadas nos testes de variação de latência, que são apresentados a seguir.

5.3.1.1 Comparação dos algoritmos usando Relógios Lógicos

Nos testes utilizando relógios lógicos, cada processo sempre pede os mesmos recursos em todas as execuções. Para esta comparação, foi utilizado o **Ambiente de testes 1**.

Analisando a Figura 5.4(a), observa-se que o procedimento de obtenção do *Control Token*, apresenta resultados semelhantes ao experimento realizado na seção anterior. Entretanto, diferencia-se no tempo de obtenção, que para esta generalização torna-se maior devido a maior quantidade de recursos espalhados no sistema, o que por conseqüência aumenta o tempo total de obtenção dos recursos. Com a variação das latências entre *clusters*, podemos notar que houve um crescimento mais acentuado no tempo da figura citada, no entanto, sendo bem menor no algoritmo proposto. Isso significa que com o aumento da latência entre *clusters*, torna-se claro o benefício da priorização das mensagens locais em relação às remotas. De fato, foi o que prejudicou o resultado obtido pelo algoritmo concorrente, que trata todos os processos indiferentemente de localização. Como um reflexo do resultado anterior, vemos também na Figura 5.4(b) que o algoritmo proposto continua sendo superior, pois ao selecionar os recursos considera a localização destes recursos, dando preferência aos recursos mais próximos.

A relação de mensagens trocadas entre processos remotos e locais, se mantém como o esperado em relação à proposta do algoritmo. Observe na Figura 5.4(d), que o número de mensagens locais trocadas pelo algoritmo proposto é cerca de 80% maior que o número de mensagens remotas. Observe também que o contrário pode ser visto nos resultados

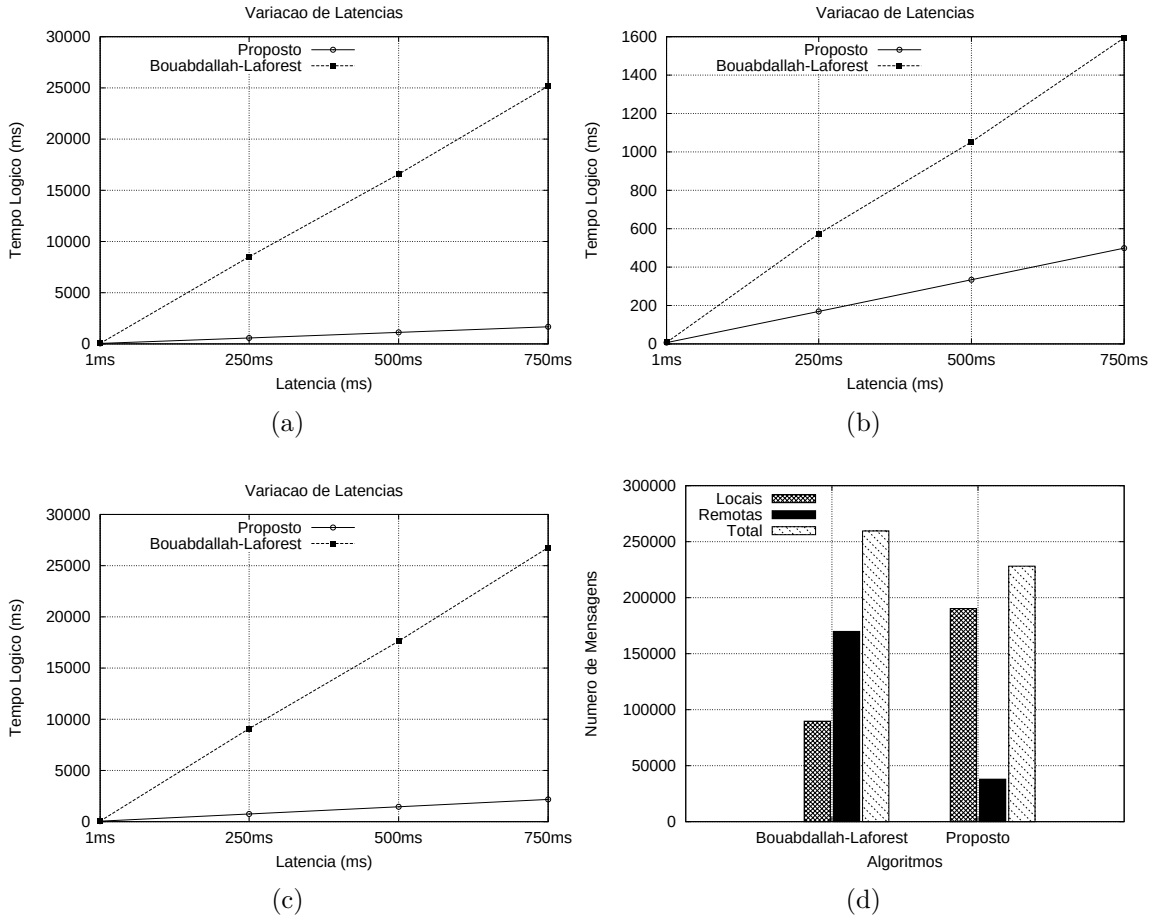

Figura 5.4: Variação da latência - Múltiplos Recursos com Múltiplas Instâncias: (a) Média do tempo lógico para obtenção do *Control Token*; (b) Média do tempo lógico para obtenção do último *token*; (c) Média da soma dos tempos anteriores, maior cadeia causal para se obter acesso à SC; e, (d) Número de mensagens trocadas.

do algoritmo de Bouabdallah-Laforest, onde é notado um aumento de 47% de mensagens remotas a mais que as mensagens locais. Como mensagens remotas são enviadas por canais com maior custo de comunicação, verificamos que quanto mais mensagens remotas são trocadas, maior o tempo de espera para obtenção da seção crítica.

Na Figura 5.4(c), é mostrada a soma das médias dos tempos lógicos para se obter acesso à seção crítica. Em virtude dos bons resultados dos testes imediatamente anteriores, essa figura reafirma os resultados superiores obtidos com o algoritmo proposto.

5.3.1.2 Comparação dos algoritmos usando Relógio Físico

Para o teste de tempo de obtenção utilizando relógio físico, também foi utilizado o **Ambiente de testes 1**.

Os resultados que podem ser visualizados na Figura 5.5, atestam que o algoritmo proposto também se manteve superior ao concorrente utilizando tempo físico como estratégia de medição, assim como observado nos resultados obtidos no teste com relógios lógicos. Como visto nos testes anteriores, o menor consumo de canais remotos no algoritmo proposto, resultou em uma curva de tempo de obtenção à seção crítica menos acentuada do que a mesma observada no algoritmo concorrente.

Assim como no teste realizado na seção anterior, usando a ferramenta NetEm, os resultados apresentados na Figura 5.6 também se aproximam bastante dos resultados obtidos com os outros testes que usam outras formas de medição de desempenho. Essencialmente, a mudança mais notável está relacionada aos tempos obtidos, que são menores visto que para este experimento foi utilizado o **Ambiente de testes 2**, que possui 24 processadores. No entanto, a estratégia da utilização da ferramenta NetEm foi de atestar que o comportamento do algoritmo se mantém similar usando-se diferentes formas de medição, o que foi conseguido.

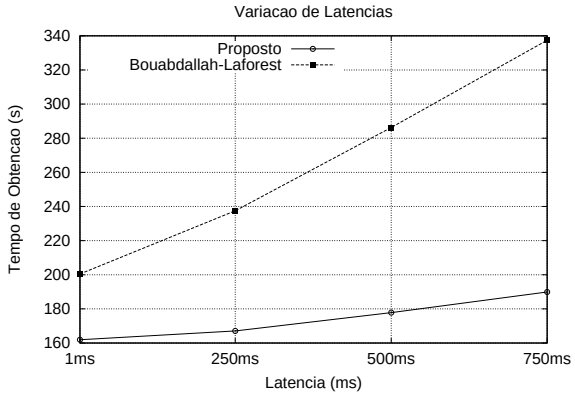

Figura 5.5: Tempo de Obtenção - Tempo Físico usando *Sleep* - Múltiplos Recursos com Múltiplas Instâncias

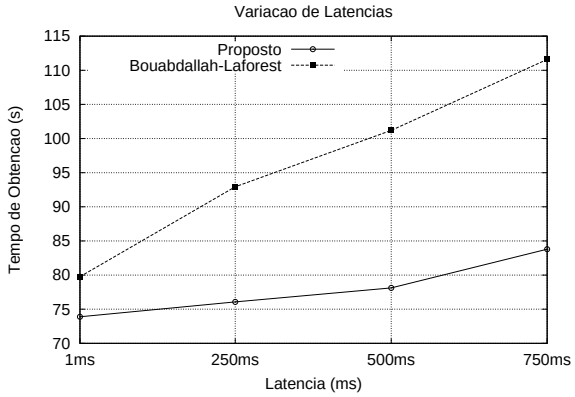

Figura 5.6: Tempo de Obtenção - Tempo Físico usando NetEm - Múltiplos Recursos com Múltiplas Instâncias

Tabela 5.3: *Tabela de Médias e Desvio Padrão* - ***Variação de Latências - Múltiplos Recursos com Múltiplas Instâncias***

Algoritmos	**Latências(ms)**	**Relógio Físico(s)**		**Relógio Lógico(ms)**	
		Média	**Desv. Pad (%)**	**Média**	**Desv. Pad (%)**
Algoritmo Proposto	1ms	161.94	4.63	50.57	0.65
	250ms	167.07	4.63	750.79	1.03
	500ms	177.79	4.63	1456.71	1.02
	750ms	189.91	4.47	2169.25	1.06
Bouabdallah Laforest	1ms	200.47	3.12	52.17	0.59
	250ms	237.47	3.27	9083.19	0.80
	500ms	286.26	4.35	17626.63	0.95
	750ms	337.44	3.90	26768.14	0.79

Tabela 5.4: *Tabela do Número de Mensagens Trocadas* - ***Variação de Latências - Múltiplos Recursos com Múltiplas Instâncias***

Algoritmos	**Latências(ms)**	**Número de Mensagens**		**Preempções**
		Remotas	**Locais**	
Algoritmo Proposto	1ms	1043.00	5276.33	88
	250ms	1053.33	5299.22	91
	500ms	1063.11	5274.00	92
	750ms	1055.22	5284.44	93
Bouabdallah Laforest	1ms	5385.78	1779.00	-
	250ms	4530.67	2694.00	-
	500ms	4462.33	2768.22	-
	750ms	4497.22	2724.11	-

Os tempos médios de obtenção e o desvio padrão, obtidos a partir da variação de latências, são apresentados na Tabela 5.3. Estes resultados mostram que os valores do tempo de obtenção usando o algoritmo Boubdallah-Laforest são muito maiores do que usando o algoritmo proposto, tanto usando relógios lógicos, quanto usando relógios físicos. Porém, utilizando relógios de tempo físico, o tempo médio de obtenção do algoritmo proposto foi em torno de 56% do algoritmo concorrente, quando a latência entre *clusters* ficou em 750ms. Observa-se que não houve uma variação notável nos valores de desvio padrão, para ambos os testes.

Detalhadamente, podemos ver o número de mensagens trocadas pelos algoritmos na Tabela 5.4. Veja que, a concentração das mensagens locais feita pelo algoritmo proposto é bem definida aqui. Isso mostra que, a estratégia adotada foi eficiente e resultou na redução do tempo de obtenção da seção crítica. O número de preempções é sumarizado nesta última tabela, onde o valor assemelha-se com o resultado obtido na primeira parte dos testes.

5.3.2 Variação do Número de Tipos de Recursos

O objetivo deste experimento é avaliar o comportamento dos algoritmos envolvidos neste trabalho, a partir da variação do número de tipos de recursos disponíveis. A quantidade de tipos de recursos variou em 01, 05, 10 e 20. Neste teste, cada processo solicitou um subconjunto aleatório destes recursos para entrar em sua seção crítica. Atribuímos também, para cada recurso, um total de 10 instâncias. Assim, um processo também pode solicitar mais de uma cópia de um mesmo tipo de recurso. Os parâmetros utilizados neste teste foram definidos como: β em 500ms, α em 500ms e a latência entre *clusters* também em 500ms. O limite de preempção foi definido com valor de 08.

Para o teste a seguir, foi utilizada somente a medição de tempo por relógio físico, sendo este, executado no **Ambiente de testes 1**.

Analisando a Figura 5.7, é possível verificar que o algoritmo proposto conseguiu reduzir o tempo de obtenção, mesmo com o aumento da variação de tipos de recursos. O ponto de maior eficiência observada foi quando a quantidade de tipos de recursos ficou entre 05 e 20, uma vez que é quando os recursos ficam mais espalhados pelo sistema. Neste caso, o algoritmo proposto conseguiu selecionar melhor estes recursos, agrupando-os em pedidos e gastando menos mensagens, além de ter dado maior prioridade a pedidos locais. Note que essa estratégia mostrou uma diferença contra o algoritmo de Bouabdallah-Laforest, que não possui uma estratégia de seleção adequada. Veja também, que mesmo quando o número recursos é igual a 01, o algoritmo proposto ainda consegue ter um resultado melhor que o concorrente.

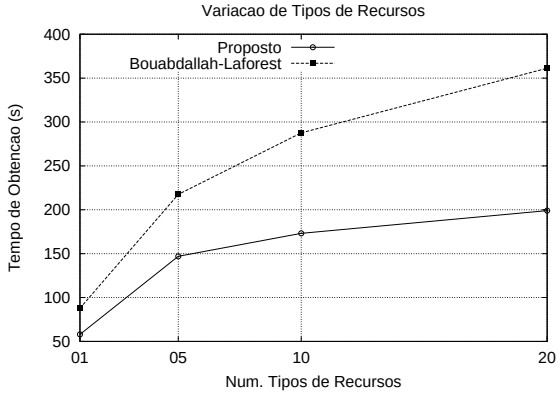

Figura 5.7: Tempo de Obtenção - Teste de variação do número de tipos de recursos - Múltiplos Recursos com Múltiplas Instâncias

Na Tabela 5.5, podemos ver os valores médios dos tempos de obtenção à seção crítica, juntamente com o desvio padrão de cada valor obtido. Note que os valores de desvio

Tabela 5.5: *Tabela de Médias e Desvio Padrão* - ***Variação do Número de Tipos de Recursos - Múltiplos Recursos com Múltiplas Instâncias***

Algoritmos	**Núm. de Tipos de Recursos**	**Tempo de Obtenção(s)**		**Núm. de Mensagens**	
		Média	**Desv. Pad (%)**	**Globais**	**Locais**
Algoritmo Proposto	01	58.04	19.19	160.40	2834.00
	05	146.97	5.49	659.40	5382.60
	10	173.17	4.75	972.80	5915.40
	20	199.02	5.00	1344.40	6387.00
Bouabdallah Laforest	01	87.86	13.97	1410.00	786.40
	05	217.56	6.19	3565.80	2025.60
	10	287.61	3.35	4572.80	2675.00
	20	361.30	3.40	5751.20	3337.40

padrão foram próximos, com exceção do valor obtido quando o número de tipos de recursos foi de apenas 01. Neste caso, como havia somente um tipo de recurso com 10 instâncias cada, esses recursos não puderam ser espalhados pelo sistema. Assim, alguns processos conseguiram obter todos seus recursos localmente, enquanto outros tiveram que realizar seus pedidos remotamente, o que gerou uma discrepância nas médias do tempo de obtenção de todas as variações, resultando em um desvio padrão um pouco mais alto.

5.3.3 Variação do Número Total de Processos

O experimento a seguir foi feito com intuito de medir a escalabilidade e a eficiência dos algoritmos a partir do crescimento do número de processos.

Neste teste, variou-se o número de processos em 12, 24, 48 e 60. Em todos os casos, os processos foram divididos igualmente entre 3 *clusters*. Assim, cada *cluster* recebeu a mesma quantidade de processos. Os parâmetros α e β, receberam ambos 500ms. A latência dos canais de comunicação que interligam os *clusters* foi fixada em 500ms. No parâmetro que limita o número de preempções, foi utilizado um valor correspondente à 50% do número de processos que compuseram cada *cluster* em cada variação. Foram utilizados 10 tipos de recursos distintos, com 10 instâncias cada.

O teste foi executado no **Ambiente de testes 1** usando apenas relógio de tempo físico.

O crescimento do número de processos, resultou em um aumento dos tempos médios para obtenção da seção crítica, como pode ser visto na Figura 5.8. Note no gráfico que ambos os algoritmos apresentaram um crescimento acentuado, entretanto, o crescimento observado na curva do gráfico que representa o algoritmo proposto, tende a ser menor que no algoritmo de Bouabdallah-Laforest, considerando-se o aumento maior do número

de processos. Analisando os resultados obtidos, podemos perceber que com o aumento do número de processos, aumenta-se também a cadeia de espera de cada processo que aguarda para obter os recursos em comum.

Os valores das médias e desvio padrão podem ser observados na Tabela 5.6, onde os valores de desvio padrão para os algoritmos não possuem grande variação.

Na Tabela 5.6, também podemos ter um resultado mais detalhado acerca da troca de mensagens. Nessa tabela, é possível ver os valores de mensagens para cada variação diferente. Neste caso, observe que à medida que o número de processos aumentou, houve também um aumento da diferença entre mensagens remotas e locais. Veja que quando foram utilizados 12 processos concorrentes, a diferença de mensagens não foi tão significativa para o algoritmo proposto. Porém, com o aumento do número de processos, observe que essa diferença foi se tornando mais evidente, pois o número de mensagens remotas foi diminuindo em relação às locais. Basicamente o contrário foi observado para o algoritmo de Bouabdallah-Laforest. O número de preempções obtido variou de acordo com o crescimento do número de processos, cujo limite de preempções também variou de acordo com o número de processos.

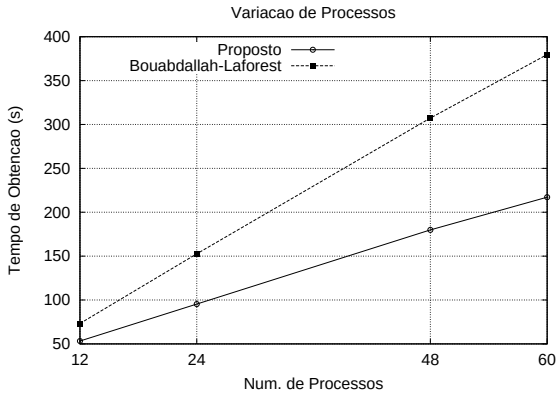

Figura 5.8: Tempo de Obtenção - Teste de variação do número de total de processos - Múltiplos Recursos com Múltiplas Instâncias

5.3.4 Variação do Número de *Clusters*

O teste a seguir, tem o objetivo de avaliar o comportamento dos algoritmos com a variação do número de *clusters* no ambiente *Grid*. O número de *clusters* variou nesse teste, em 01, 03, 06 e 12. Nessa variação, cada *cluster* recebeu partes iguais do número total de processadores, no caso 48. Os parâmetros utilizados neste teste foram definidos como: β em 500ms, α em 500ms e a latência entre *clusters* também em 500ms. No parâmetro

Tabela 5.6: *Tabela de Médias e Desvio Padrão* - ***Variação do Número de Processos - Múltiplos Recursos com Múltiplas Instâncias***

Algoritmos	**Núm. de Processos**	**Tempo de Obtenção(s)**		**Núm. de Mensagens**		**Preempções**
		Média	**Desv. Pad (%)**	**Globais**	**Locais**	
Algoritmo Proposto	12	53.27	2.79	690.00	846.00	50
	24	95.36	4.57	869.00	2296.00	52
	48	179.88	3.55	1027.00	5368.00	95
	60	217.12	4.64	1021.00	6866.00	120
Bouabdallah Laforest	12	73.26	2.51	1329.00	376.00	-
	24	152.65	2.63	2787.00	794.00	-
	48	307.35	2.70	5647.00	1648.00	-
	60	379.66	2.69	6854.00	2287.00	-

de limite de preempções, foi atribuído um valor correspondente à 50% do número de processos que compuseram cada *cluster* em cada variação. Foram utilizados 10 tipos de recursos com 10 instâncias cada.

O ambiente de testes utilizado, foi o **Ambiente de testes 1**, também foi utilizado apenas o relógio de tempo físico para a medição de desempenho.

O crescimento do número de *clusters*, faz com que a quantidade de processos distribuídos para cada *cluster* seja menor. Assim, esse ambiente obriga os processos a trocarem um maior número de mensagens remotas, pois neste caso, muitos pedidos são feitos para processos remotos. O resultado deste teste pode ser visto na Figura 5.9. Observe que o algoritmo de Bouabdallah-Laforest foi consideravelmente mais ineficiente do que o algoritmo proposto, este último que manteve um tempo linear enquanto o algoritmo comparado chegou a dobrar o tempo de obtenção, no teste usando 12 *clusters*. Isso se deu, devido ao fato do algoritmo de Bouabdallah-Laforest não fazer distinção entre processos remotos ou locais. Além disso, a etapa de inicialização deste algoritmo obriga todos os processos a solicitarem o *Control Token* sempre a um mesmo processo, com o aumento do número de *clusters* e conseqüentemente o número de canais remotos, esse procedimento gera um *overhead* alto na comunicação de todos os processos para este único processo. Já no caso do algoritmo proposto, é possível ver uma clara estabilidade nas médias dos tempos de obtenção. Isso foi possível, graças às estratégias adotadas tanto na etapa de obtenção do *Control Token*, quanto na etapa de obtenção dos recursos. A etapa de inicialização utilizada no algoritmo proposto também foi importante, pois faz com que os processos sempre façam pedidos inicialmente a um nó líder, que é local. Além disso, na etapa de pedidos de recursos, é feita uma seleção dos recursos, onde recursos locais possuem maior preferência.

Na Tabela 5.7, podemos ver os valores do resultado obtido na figura anterior. Ob-

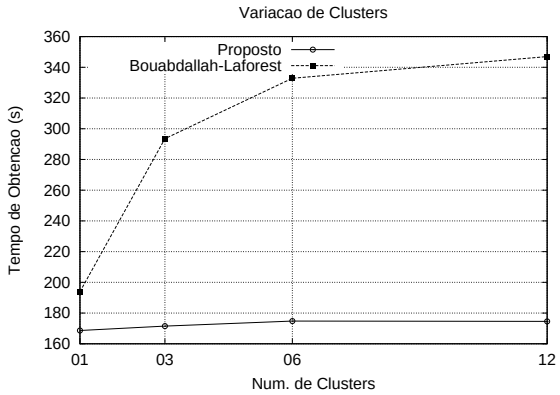

Figura 5.9: Tempo de Obtenção - Teste de variação do número de *clusters* - Múltiplos Recursos com Múltiplas Instâncias

Tabela 5.7: *Tabela de Médias e Desvio Padrão* - ***Variação do Número de Clusters - Múltiplos Recursos com Múltiplas Instâncias***

Algoritmos	**Núm. de Clusters**	**Tempo de Obtenção(s)**		**Núm. de Mensagens**		**Preempções**
		Média	**Desv. Pad (%)**	**Globais**	**Locais**	
Algoritmo Proposto	01	168.62	3.54	0.00	8137.25	0.00
	03	171.50	4.54	961.75	6029.75	92.75
	06	174.77	13.41	1674.25	5162.00	151.00
	12	174.58	12.54	2304.75	4136.00	214.75
Bouabdallah Laforest	01	193.89	3.28	0.00	7192.75	-
	03	293.55	3.42	4815.00	2399.50	-
	06	332.84	3.18	6347.00	905.75	-
	12	346.96	3.04	6923.00	302.75	-

serve que, quando se utilizou apenas 01 *cluster*, não houve nenhuma preempção visto que também não havia nenhum processo remoto. Por outro lado, quando o número de *cluster* foi de 12, o número de preempções foi mais que o dobro em relação ao teste com 3 *cluster*. Note também que os valores de desvio padrão para o algoritmo proposto ficaram um pouco mais altos, quando o número de *clusters* foi de 06 e 12. Com o aumento do número de *clusters*, aumentou-se também o número de canais remotos, e conseqüentemente, o custo da cadeia de espera para se obter acesso à seção crítica. Isso fez com que houvesse uma variação maior nas médias dos tempos de obtenção, resultando em um desvio padrão maior para estes valores.

Capítulo 6

Conclusões e Trabalhos Futuros

O trabalho teve como objetivo propor um algoritmo para resolver o problema da alocação de múltiplos recursos com múltiplas instâncias, que considerasse a heterogeneidade de latência na comunicação em um ambiente de *Grid*. O algoritmo proposto foi comparado com o algoritmo de Bouabdallah-Laforest [9].

Para a avaliação destes algoritmos, também foi proposto um relógio lógico utilizado para medir o tempo de espera para um processo obter o *Control Token* e também os *tokens* dos recursos, e conseqüentemente entrar em sua seção crítica. Tal relógio lógico considera as diferenças de latências nos canais de comunicação. O modelo de relógio lógico utilizado também mostrou ser muito útil, e pode ser empregado na avaliação de desempenho em diversos outros tipos de experimentos.

Os testes foram repetidos considerando também o tempo medido utilizando relógio de parede, sendo que para isso, foram utilizadas duas estratégias diferentes para emular uma infra-estrutura de *Grid*.

Os resultados obtidos por estas diferentes formas de avaliação foram bastante semelhantes, e atestou que o algoritmo proposto é mais eficiente do que o algoritmo de Bouabdallah-Laforest [9] utilizado na comparação.

Por fim, a redução das mensagens entre clusters e do tempo de espera para entrar na seção crítica mostrou que o algoritmo proposto é bastante atraente para ambientes heterogêneos tais como *Grids* Computacionais.

Para trabalhos futuros, sugerimos uma adaptação do algoritmo proposto adicionando um mecanismo de tolerância a falhas, utilizando alguns conceitos apresentados em [53, 54, 3], os quais acreditamos poder ser facilmente incorporados ao algoritmo proposto de-

vido à proximidade das técnicas utilizadas por eles. Uma outra sugestão é a utilização de algoritmos mais eficientes na seleção dos recursos em conjunto com o algoritmo proposto. Observamos que no momento no qual o processo obtém a estrutura do *Control Token*, ele possui ali todas as informações necessárias para uma escolha mais otimizada dos recursos. Além disso, algoritmos especializados em seleção de recursos também podem utilizar outros critérios para uma seleção mais adequada desses recursos, como por exemplo o tempo de utilização daquele recurso.

Referências

[1] `http://www.linux-foundation.org/en/Net:Netem`. [Acesso feito em: 2008.03.19].

[2] `http://www.linux-foundation.org/en/Net:Iproute2`. [Acesso feito em: 2008.03.19].

[3] Agrawal, D. e Abbadi, A. E. A token-based fault-tolerant distributed mutual exclusion algorithm. *Journal of Parallel and Distributed Computing 24*, 2 (1995), 164–176.

[4] Ahamad, M., Ammar, M. H. e Cheung, S. Y. Multidimensional voting. *ACM Transactions on Computer Systems 9*, 4 (1991), 399–431.

[5] Barbara, D., Garcia-Molina, H. e Spauster, A. Increasing availability under mutual exclusion constraints with dynamic vote reassignment. *ACM Transactions on Computer Systems 7*, 4 (1989), 394–426.

[6] Barbosa, V. C. *An introduction to distributed algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[7] Bertier, M., Arantes, L. e Sens, P. Hierarchical token based mutual exclusion algorithms. Em *CCGRID'04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 539–546.

[8] Bertier, M., Arantes, L. e Sens, P. Distributed mutual exclusion algorithms for grid applications: a hierarchical approach. *JPDC: Journal of Parallel and Distributed Computing 66*, 1 (2006), 128–144.

[9] Bouabdallah, A. e Laforest, C. A distributed token-based algorithm for the dynamic resource allocation problem. *SIGOPS Operating Systems Review 34*, 3 (2000), 60–68.

[10] Bulgannawar, S. e Vaidya, N. H. A distributed k-mutual exclusion algorithm. Em *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 153–160.

[11] Cantarell, S., Datta, A. K., Petit, F. e Villain, V. Token based group mutual exclusion for asynchronous rings (extended abstract). Em *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems* (Washington, DC, USA, 2001), IEEE Computer Society, p. 691.

[12] CERN. Large hadron collider, global grid service for lhc computing succeeds in gigabyte-per-second challenge. `http://press.web.cern.ch/Press/PressReleases/Releases2006/PR03.06E.html`, 2006. [Acesso feito em: 2007.11.07].

[13] Chang, Y.-I. A simulation study on distributed mutual exclusion. *JPDC: Journal of Parallel and Distributed Computing 33*, 2 (1996), 107–121.

[14] Chang, Y.-I., Singhal, M. e Liu, M. A hybrid approach to mutual exclusion for distributed systems. *Computer Software and Applications Conference, COMPSAC 90* (1990), 289–294.

[15] Chang, Y.-I., Singhal, M. e Liu, M. T. A fault tolerant algorithm for distributed mutual exclusion. Em *Proceedings Ninth Symposium on Reliable Distributed Systems (9th SRDS'90)* (Huntsville, Alabama, USA, outubro de 1990), IEEE Computer Society Press, pp. 146–154.

[16] Chang, Y.-I., Singhal, M. e Liu, M. T. An improved $O(logn)$ mutual exclusion algorithm for distributed systems. Em *ICPP International Conference on Parallel Processing* (1990), pp. 295–302.

[17] DeMent, N. S. e Srimani, P. K. A new algorithm for mutual exclusions in distributed systems. *Journal of Systems and Software 26*, 2 (1994), 179–191.

[18] Drummond, L. M. A. e Barbosa, V. C. On reducing the complexity of matrix clocks. *Parallel Computing 29*, 7 (julho de 2003), 895–905.

[19] Erciyes, K. Distributed mutual exclusion algorithms on a ring of clusters. Em *Computational Science and Its Applications - ICCSA 2004, International Conference* (2004), A. Laganà, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, e O. Gervasi, Eds., vol. 3045 of *Lecture Notes in Computer Science*, Springer, pp. 518–527.

[20] Fidge, C. J. Timestamps in message-passing systems that preserve the partial ordering. Em *Proc. 11th Australian Computer Science Conference* (1988), pp. 56–66.

[21] Garcia-Molina, H. e Barbara, D. How to assign votes in a distributed system. *JACM: Journal of the ACM 32*, 4 (1985), 841–860.

[22] Gifford, D. K. Weighted voting for replicated data. Em *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles* (New York, NY, USA, 1979), ACM, pp. 150–162.

[23] Han, H., Jung, H., Yeom, H. Y., Kweon, H. S. e Lee, J. HVEM grid: Experiences in constructing an electron microscopy grid. Em *Frontiers of WWW Research and Development - APWeb 2006, 8th Asia-Pacific Web Conference, Harbin, China, January 16-18, 2006, Proceedings* (2006), X. Zhou, J. Li, H. T. Shen, M. Kitsuregawa, e Y. Zhang, Eds., vol. 3841 of *Lecture Notes in Computer Science*, Springer, pp. 1159–1162.

[24] Helary, J. M., Plouzeau, N. e Raynal, M. A distributed algorithm for mutual exclusion in an arbitrary network. *The Computer Journal. 31*, 4 (1988), 289–295.

[25] Hemminger, S. Network emulation with netem. Tech report, Open Source Development lab, 4 2005. [Acesso feito em: 2008.03.19].

[26] Housni, A. e Trehel, M. Distributed mutual exclusion token-permission based by prioritized groups. *Computer Systems and Applications, ACS/IEEE International Conference* (2001), 253–259.

[27] Jajodia, S. e Mutchler, D. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems 15*, 2 (1990), 230–280.

[28] Kakugawa, H. A Study on Distributed k-Mutual Exclusion Algorithms. Tese de Mestrado, Hiroshima University, Feb. 1995.

[29] Kakugawa, H., Fujita, S., Yamashita, M. e Ae, T. A distributed k-mutual exclusion algorithm using k-coterie. *Information Processing Letters 49*, 4 (1994), 213–218.

[30] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[31] Lann, G. L. Distributed systems – toward a formal approach. Em *Proceedings of the IFIP Congress 77* (1977), pp. 155–160.

[32] Liu, C., Yang, L., Foster, I. e Angulo, D. Design and evaluation of a resource selection framework for grid applications. Em *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 63–72.

[33] Maddi, A. Token based solutions to M resources allocation problem. Em *SAC '97: Proceedings of the 1997 ACM symposium on Applied computing* (New York, NY, USA, 1997), ACM, pp. 340–344.

[34] Maekawa, M. A $O\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems 3*, 2 (1985), 145–159.

[35] Mattern, F. Virtual time and global states of distributed systems. Em *Proc. Workshop on Parallel and Distributed Algorithms* (North-Holland / Elsevier, 1989), C. M. et al., Ed., pp. 215–226. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[36] Naimi, M. Distributed algorithm for k-entries to critical section based on the directed graphs. *SIGOPS Operating Systems Review 27*, 4 (1993), 67–75.

[37] Naimi, M. e Trehel, M. An improvement of the log(n) distributed algorithm for mutual exclusion. Em *ICDCS: 7th International Conference on Distributed Computing Systems* (Berlin, Germany, 1987), pp. 156–172.

[38] Naimi, M., Trehel, M. e Arnold, A. A log (n) distributed mutual exclusion algorithm based on path reversal. *JPDC: Journal of Parallel and Distributed Computing 34*, 1 (1996), 1–13.

[39] Nassif, L. N., Nogueira, J. M. e de Andrade, F. V. Distributed resource selection in grid using decision theory. Em *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 327–334.

[40] Neilsen, M. L. e Mizuno, M. A dag-based algorithm for distributed mutual exclusion. Em *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)* (Washington, DC, 1991), IEEE Computer Society, pp. 354–360.

[41] Nielsen, M. L. e Mizuno, M. Coterie join algorithm. *IEEE Transactions on Parallel and Distributed Systems 3*, 5 (1992), 582–590.

[42] Nishio, S., Li, K. e Manning, E. A resilient mutual exclusion algorithm for computer networks. *IEEE Transactions on Parallel and Distributed Systems 01*, 3 (1990), 344–356.

[43] Raymond, K. A distributed algorithm for multiple entries to a critical section. *IPL: Information Processing Letters 30*, 4 (1989), 189–193.

[44] Raymond, K. A distributed algorithm for multiple entries to a critical section. *IPL: Information Processing Letters 30*, 4 (1989), 189–193.

[45] Raymond, K. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems 7*, 1 (1989), 61–77.

[46] Raynal, M. A distributed solution to the k-out of-M resources allocation problem. *Lecture Notes in Computer Science 497* (1991), 599–609.

[47] Raynal, M. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Operating Systems Review 25*, 2 (1991), 47–50.

[48] Ricart, G. e Agrawala, A. K. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM 24*, 1 (1981), 9–17.

[49] Saxena, P. C. e Rai, J. A survey of permission-based distributed mutual exclusion algorithms. *Computer Standards & Interfaces 25*, 2 (2003), 159–181.

[50] Singhal, M. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers 38*, 5 (1989), 651–662.

[51] Singhal, M. A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems 3*, 1 (1992), 121–125.

[52] Singhal, M. A taxonomy of distributed mutual exclusion. *JPDC: Journal of Parallel and Distributed Computing 18*, 1 (1993), 94–101.

[53] Sopena, J., Arantes, L. e Sens, P. Performance evaluation of a fair fault-tolerant mutual exclusion algorithm. Em *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 225–234.

[54] Sopena, J., Arantes, L. B., Bertier, M. e Sens, P. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. Em *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference* (Lisbon, Portugal, agosto de-setembro de 2005), J. C. Cunha e P. D. Medeiros, Eds., vol. 3648 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag (New York), pp. 654–663.

[55] Sopena, J., Legond-Aubry, F., Arantes, L. e Sens, P. A composition approach to mutual exclusion algorithms for grid applications. Em *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing* (Washington, DC, USA, 2007), IEEE Computer Society, p. 65.

[56] Srimani, P.K.; Reddy, R. A new algorithm for simultaneous multiple entries in critical section in a distributed system. Em *Computers and Communications, 1990. Conference Proceedings., Ninth Annual International Phoenix Conference on* (1990), p. 895.

[57] Suzuki, I. e Kasami, T. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems 3*, 4 (1985), 344–349.

[58] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems 4*, 2 (1979), 180–209.

[59] van de Snepscheut, J. L. A. Fair mutual exclusion on a graph of processes. *Distributed Computing 2*, 2 (1987), 113–115.

[60] Wang, S. e Lang, S. D. A tree-based distributed algorithm for the K-entry critical section problem. Em *ICPADS: Proceedings 1994 International Conference on Parallel and Distributed Systems* (1994), L. M. Ni, Ed., IEEE Computer Society, pp. 592–599.

[61] Yan, Y., Zhang, X. e Yang, H. A fast token-chasing mutual exclusion algorithm in arbitrary network topologies. *JPDC: Journal of Parallel and Distributed Computing 35*, 2 (1996), 156–172.