Um Algoritmo Eficiente para Exclusão Mútua Distribuída

Guilherme Pádua Teixeira* Kêmio de Oliveira Couto†
Marco Aurélio de Souza Mendes‡ Osvaldo S. F. Carvalho§

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Caixa Postal 702, CEP 30.161-920
Belo Horizonte, MG, Brasil

Resumo

Este trabalho apresenta um novo algoritmo para exclusão mútua distribuída. O algoritmo aqui proposto é baseado nas idéias de grafos acíclicos de Chandy and Misra, do *token* com muitas informações de Suzuki-Kazami e na idéia do *holder* de Raymond. O número de mensagens trocadas por seção crítica é O(log N) em carga baixa, onde N é o número de nodos da rede. Na situação de saturação, entretanto, somente duas mensagens são necessárias por invocação de seção crítica. É também proposto aqui um modelo de simulação para algoritmos distribuídos de exclusão mútua. Baseado nele, uma comparação é realizada entre vários algoritmos propostos na literatura e o nosso.

Abstract

This work presents a new algorithm for distributed mutual exclusion. The algorithm is based on the ideas of the acyclic graph of Chandy and Misra, the token with many information of Suzuki and Kazami and on the idea of the holder of Raymod . The number of messages exchanged per critical section is O(log N) under light demand, where N is the number of nodes of the network. In the situation of saturation, however, only two messages are required per critical section invocation. It is also proposed in the paper a simulation model for distributed mutual exclusion algorithms. Based on it, a comparison is performed among the major algorithms proposed in the literature and ours.

*Graduado em Ciência da Computação pelo DCC/UFMG - e-mail: guigo@dcc.ufmg.br

†Aluno de pós-graduação do DCC/UFMG - e-mail: chalub@dcc.ufmg.br

‡Aluno de pós-graduação do DCC/UFMG - e-mail: corelio@dcc.ufmg.br

§Professor do DCC/UFMG - e-mail: vado@dcc.ufmg.br

1 Introdução

Sistemas distribuídos são um meio adequado para o tratamento de muitos problemas em ciência da computação. No entanto, incertezas nos atrasos de mensagens e na velocidade relativa que os *jobs* são executados tornam a sincronização entre processos uma tarefa complicada. A exclusão mútua entre processos é uma importante ferramenta para tratar este tipo de problema.

Muitos algoritmos para exclusão mútua distribuída são propostos na literatura. Inicialmente, Lamport [Lam78] propôs um algoritmo que requer $3 * (N - 1)$ mensagens por invocação da seção crítica, onde N é o número de nodos presentes na rede. Desde então, o número de mensagens trocadas tem diminuído com o advento de novos algoritmos. Atualmente, os algoritmos dessa classe de problema são de ordem logarítmica [Ray89] [AAH89]. Além do número de mensagens trocadas outras métricas devem ser avaliadas, tais como tempo de resposta e atraso de sincronização.

Este artigo apresenta um novo algoritmo para exclusão mútua distribuída em redes de computadores que troca um número igual ou menor de mensagens que qualquer outro algoritmo por nós observado. O algoritmo é *token-based* e utiliza um esquema de compressão de caminho similar ao visto em [AAH89]. Assume-se que um nodo pode se comunicar com qualquer outro nodo da rede através de mensagens e não há memória compartilhada. O tempo de tráfego das mensagens é indeterminado mas finito e não há falhas na comunicação. Além disso, presume-se que não haja falhas dos nodos, i.e, os nodos são completamente confiáveis.

O algoritmo proposto neste artigo é comparado com outros citados na literatura. Os algoritmos foram implementados e testados através de um ambiente de simulação de algoritmos distribuídos[CMC94]. Esse ambiente dá flexibilidade para a especificação de um sistema distribuído e coleta as informaçõs mais relevantes para comparação entre algoritmos.

No decorrer deste artigo, o novo algoritmo será apresentado, juntamente com garantias de exclusão mútua e ausência de *deadlock* e *starvation* e uma análise de complexidade do mesmo. Serão descritos então o modelo de comparação usado e apresentados os resultados das simulações e uma análise das mesmas.

2 A Fila de Precedências

Em [CM84], o problema de conflitos e precedências para alocação de recursos é mapeada para grafos dirigidos onde:

- nodos são sítios;
- existem arestas se houver conflito entre os sítios; e
- o sentido da aresta é do sítio de maior para o de menor precedência.

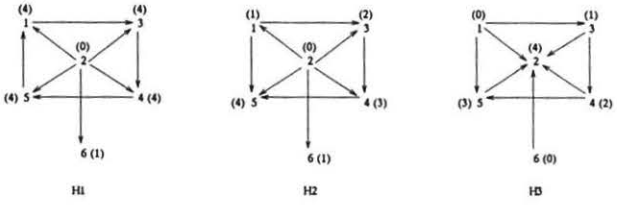

Figura 1: Grafos de Precedência

Considere a figura 1. No grafo H1, os nodos 1 e 4 não estão em conflito, mas ambos concorrem por recursos com os nodos 2, 3 e 5. É interessante notar um ciclo de precedências entre os nodos 1, 3, 4 e 5, onde 1 precede 3, que precede 4, que precede 5, que, por sua vez, precede o 1. No grafo H2 isso já não ocorre.

Um conceito extraído do grafo é o de profundidade: os nodos que não têm arestas incidentes são ditos ter profundidade zero; e a profundidade dos demais é o número máximo de arestas desde um nodo que tenha profundidade zero até ele.

É interessante notar que num grafo acíclico sempre haverá um nodo de profundidade zero e as profundidades entre os nodos vizinhos serão sempre diferentes. A profundidade pode ser usada então para resolver os conflitos entre vizinhos sem que haja *deadlocks* na rede. Além disso, se o sentido de todos as arestas incidentes a um nodo for invertido, de modo atômico, continua-se com um grafo acíclico (grafo H3), e portanto sem *starvation*.

Num regime de exclusão mútua, todos os nodos entram em conflito entre si por um único recurso, portanto pode-se usar um grafo completo no qual, para utilizar o recurso, um nodo deve preceder todos os outros.

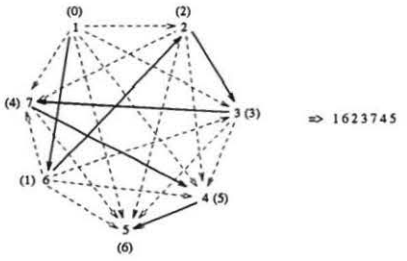

Figura 2: Grafo de Precedência Completo

Na figura 2 o nodo 1 tem profundidade zero. O nodo 6 tem profundidade 1, pois só perde o conflito para o nodo 1. O nodo 2 tem profundidade 2, pois perde para os nodos 1 e 6, que têm profundidade 0 e 1 respectivamente. Continuando assim até o menos prioritário observa-se uma ordem crescente de profundidade. Essa ordem crescente pode ser vista como uma ordem de precedências global e pode ser representada por uma fila de prioridades. É possível, então, decidir qualquer conflito entre os sítios através da fila de precedências, ou seja, *no deadlocks*.

Quando o sentido de todos os arcos incidentes a um nodo muda, a profundidade do nodo torna-se máxima; ou seja, ele vai para o fim da fila de prioridades. Essa ação indica que se dois nodos i e j disputaram o recurso uma vez e o nodo i ganhou, na próxima vez que disputarem antes do j conseguir utilizar, o nodo j irá ganhar. Isso garante *no starvation*.

No algoritmo aqui proposto, a fila de prioridades é guardada no Token e só quem o detém pode utilizar o recurso e decidir quem será o próximo; isto é, a resolução de conflitos é feita por apenas um nodo e com base na fila de prioridades. Os demais nodos têm a informação de um caminho que leva ao nodo detentor do Token. Ao terminar de utilizar o recurso, o nodo envia o Token para o nodo de demanda mais prioritária e passa a ter a menor prioridade da fila.

3 A Descrição do Algoritmo

Para viabilizar a idéia, cada nodo guarda algumas informações, o token transporta outras e os nodos trocam dois tipos de mensagens: Request e Token.

- Informações armazenadas no nodo:
  - **owner**, cujo valor indica o nodo que está supostamente com o token. Se o valor for 0 indica que o nodo já solicitou o recurso compartilhado, e se for o valor do próprio nodo indica que o nodo está com o token;
  - **RequestList**, lista de nodos que mandaram pedidos para o nodo enquanto este esperava pelo token ou estava em sua seção crítica. Um nodo passa a esperar o token quando pede o recurso. Essa lista de pedidos será anexada para a lista acumulada que vem com o token (ordenada em função da lista de precedência);

- Informações guardadas no token:
  - **RequestQueue**, fila que guarda os pedidos acumulados dos nodos visitados. A cada utilização do recurso, a RequestList do nodo que acabou de utilizar é inserida na RequestQueue do token.
  - **PriorityQueue**, fila que indica as prioridades de todos os nodos da rede.
- Estados dos nodos:
  - **Thinking:** indica inatividade do nodo em relação à solicitação do recurso.
  - **Hungry:** indica que o nodo está esperando o token solicitado.
  - **Eating:** indica que o nodo está utilizando o recurso.

Abaixo é dada uma descrição mais formal do algoritmo.

R 1 Ficando com fome:

$$Thinking \rightarrow$$
$$\quad Hungry$$

R 2 Passando a comer:

$$Hungry \wedge owner = Itself \rightarrow$$
$$\quad Eating$$

R 3 Pedindo o Token:

$$Hungry \wedge owner \neq Itself \wedge owner \neq 0 \rightarrow$$
$$\quad Send(Request(Itself), owner)$$
$$\quad owner := 0$$

R 4 Recebendo um Request:

$$Receive(a, Request) \rightarrow$$
$$\quad Eating \vee owner = 0 \rightarrow$$
$$\quad\quad RequestList.Insert(Request.RequestingNode)$$
$$\quad owner \neq Itself \wedge owner \neq 0 \rightarrow$$
$$\quad\quad Send(Request(Request.RequestingNode), owner)$$
$$\quad\quad owner := Request.RequestingNode$$
$$\quad Thinking \wedge owner = Itself \rightarrow$$
$$\quad\quad RequestQueue.Insert(Request.RequestingNode)$$

R 5 Recebendo o Token:

$$Receive(a, Token) \rightarrow$$
$$\quad owner := Itself$$

R 6 Enviando o Token:

$$Thinking \wedge owner = Itself \wedge \sim RequestQueue.Empty() \rightarrow$$
$$\quad owner := RequestQueue.LowestPriorityRequest()$$
$$\quad Send(Token, RequestQueue.HighestPriorityRequest())$$

R 7 Passando a pensar:

Eating →
Thinking
RequestQueue.RemoveRequest(Itself)
PriorityQueue.LeastPriority(Itself)
RequestQueue.Merge(RequestList, PriorityQueue)
RequestList.Clear()

A função *RequestQueue.Merge(RequestList, PriorityQueue)* apenda a RequestList na RequestQueue ordenando pela PriorityQueue. As demais funções usadas tem significados diretos e não são, portanto, explicadas aqui. Por *Requesting Node*, entenda-se o nodo que originalmente fez o pedido do recurso compartilhado.

A inicialização do algoritmo é dada pela escolha de um nodo como nodo privilegiado. Este nodo deterá o token, e os valores de suas variáveis locais serão:

```
RequestList := Empty List
Owner := Itself
```

A inicialização para todos os outros nodos é dada abaixo:

```
RequestList := Empty List
Owner := Privileged Node
```

4 Garantia de Correção

A decisão de quem será o próximo nodo a utilizar o recurso é centralizada no detentor do token. Essa decisão é tomada com base nos pedidos acumulados no token através da RequestQueue. Por ser centralizada em um único nodo, garante a exclusão mútua do recurso. Como quem decide faz isso pela fila de prioridades, a ausência de *deadlocks* é garantida. Quando um nodo deixa sua seção crítica, ele coloca a si próprio no fim da fila de prioridades, o que garante a ausência de *starvation*. Tudo isso têm como base a unicidade do token.

Basta agora garantir que qualquer nodo vai conseguir fazer seu pedido chegar ao token num tempo finito. Para isso, suponha que um nodo i envia um pedido ao seu owner (nodo j), o que pode resultar em três situações:

- O nodo j tem o token: então o pedido chegou ao token;
- O nodo j não detém o token mas seu owner é diferente de zero; então j repassa o pedido de i para o seu owner;
- O nodo j não detém o token e não sabe onde este está (owner é igual a zero); isso ocorre quando o nodo j fez pedido.

No primeiro caso a prova está completa visto que o pedido do nodo i alcançou o token. O segundo caso pode ser reduzido ao problema original, dado que se deve garantir, agora, que o pedido de i repassado pelo nodo j chegue ao token. O último caso também se reduz ao problema original, visto que deve-se garantir que o pedido de j chegue ao token. Se o pedido do nodo j chegar ao token em um tempo finito o token alcançará j em um tempo também finito, o que implica que o pedido de i alcançou o token.

Dado que um pedido propagado nunca volta para o nodo que o originou, o segundo e o último casos ocorrem no máximo $n-1$ vezes. Então, um pedido originado em i sempre alcança o token em um tempo finito.

5 Análise de Complexidade

5.1 A Árvore Dinâmica de Execução

Para uma melhor análise do algoritmo será associada uma árvore a um estado de sua execução e algumas mudanças que ocorrem nessa árvore, quando algum passo do algoritmo é executado, serão avaliadas.

Os nodos da árvore serão os sítios e as arestas existirão entre dois nodos i e j quando:

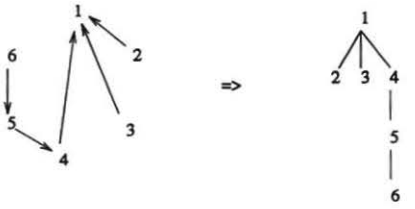

Figura 3: A Árvove Dinâmica de Execução

1. o owner de i for o nodo j;

2. i enviou um pedido para j que está em trânsito sobre a rede;

3. um pedido originado em i está na RequestList de j;

4. um pedido originado em i alcançou o token. O token está em j ou trafegando em sua direção.

A *raiz primária* será o nodo que detém o token. Haverá uma cadeia entre i e j se há um caminho entre i e j onde os arcos são sempre como em 1. Cada nodo i cujo $owner_i = 0$ é chamado *raiz secundária*. A figura 3 apresenta uma árvore com raiz primária somente.

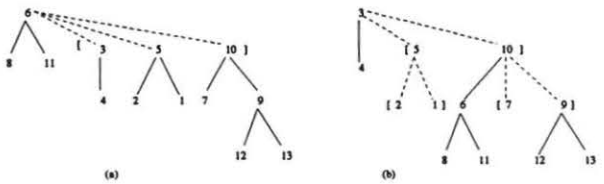

Figura 4: A Árvove Dinâmica de Execução

Um nodo após solicitar o recurso será uma raiz secundária. Este nodo não propaga os pedidos recebidos de seus filhos. Ele aguarda até que o token o alcance. Como exemplos temos a figura 4(a), no qual os nodos 3, 5 e 10 fazem parte da RequestQueue do token; e a figura 4(b), no qual os nodos 5 e 10 fazem parte da RequestQueue do token, os nodos 2 e 1 fazem parte da RequestList do nodo 5 e os nodos 7 e 9 da RequestList do nodo 10, havendo portanto várias raízes secundárias. Por essas observações, percebe-se que quando a RequestList e a RequestQueue não estão vazias o número de mensagens por demanda tende a cair.

Quando um nodo interno da árvore solicita o recurso todos os nodos que propagam seu pedido passam a ser seus filhos diretos e suas sub-árvores os seguem. Quando um nodo libera o recurso, todos os nodos de sua sub-árvore se tornam descendentes do nodo menos prioritário da RequestQueue. Quando um nodo entra em sua seção crítica, os nodos que se encontram em sua RequestList passam a fazer parte da RequestQueue do token e este nodo passa, então, a ser a raiz primária.

Na sequência mostrada pela figura 5 pode-se notar dois momentos distintos: uma situação de carga alta, ou seja, ocorre um número maior de pedidos que liberações, e isso resulta em uma árvore com tamanho médio das cadeias bastante pequeno; outra de carga baixa, onde ocorre o contrário e a profundidade média aumenta. Com isso percebe-se, sem nenhuma análise mais profunda, que o algoritmo apresenta uma melhor performance em situações de carga alta ou saturação do sistema. E, a medida que a carga diminui, as cadeias passam a ter maior profundidade, o que penaliza a performance.

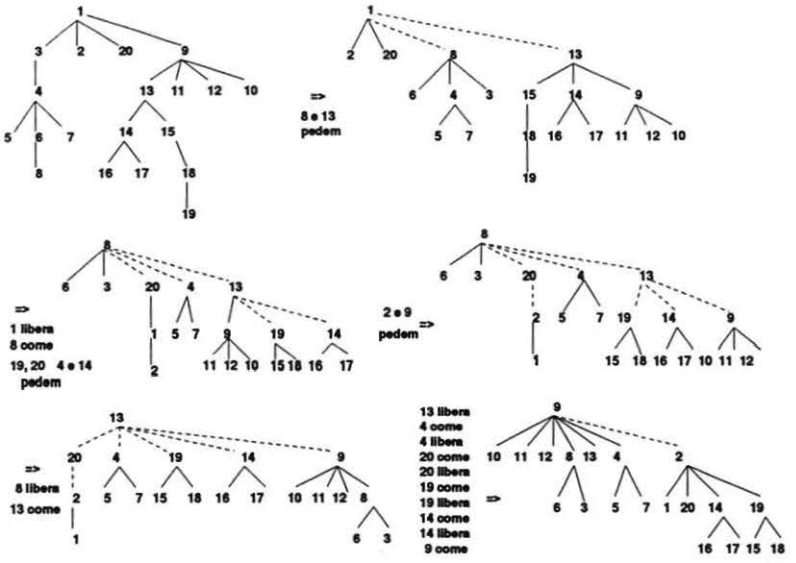

Figura 5: Transições numa Árvove Dinâmica de Execução

5.2 Análise dos Estados Limites

Os seguintes limites serão considerados: sistema saturado (ambiente em carga altíssima) e sistema ocioso (ambiente em carga baixíssima). Um estado de saturação ocorre quando qualquer nodo, ao liberar o recurso, o pede novamente, antes mesmo que a seção crítica seguinte tenha início. Assim, uma fotografia dos nodos com o sistema neste estado teria um nodo utilizando o recurso compartilhado e os restantes esperando para utilizar. Quando um nodo pede o recurso e nenhum outro o está utilizando e nem deseja fazê-lo, dizemos que o sistema se encontra em estado ocioso.

Figura 6: Sistema saturado

Na figura 6 (sistema saturado), cada nodo envia apenas uma mensagem para solicitar o token, pois todos os outros nodos são raízes. Para receber a permissão mais uma mensagem é necessária. Assim temos duas mensagens trocadas para uma demanda da seção crítica em um sistema saturado. Esse resultado, portanto, independe do número de nodos do sistema.

Em um sistema ocioso, a árvore pode assumir as mais variadas formas, considerando as demandas aleatórias. Como exemplo é detalhada a árvore de execução para uma demanda totalmente equilibrada, na qual um nodo que acabou de liberar o recurso pedi-lo-á novamente tempo suficiente depois de todos

os outros nodos o utilizarem, o que gera uma situação onde há somente a raiz primária e uma cadeia com profundidade máxima (figura 7).

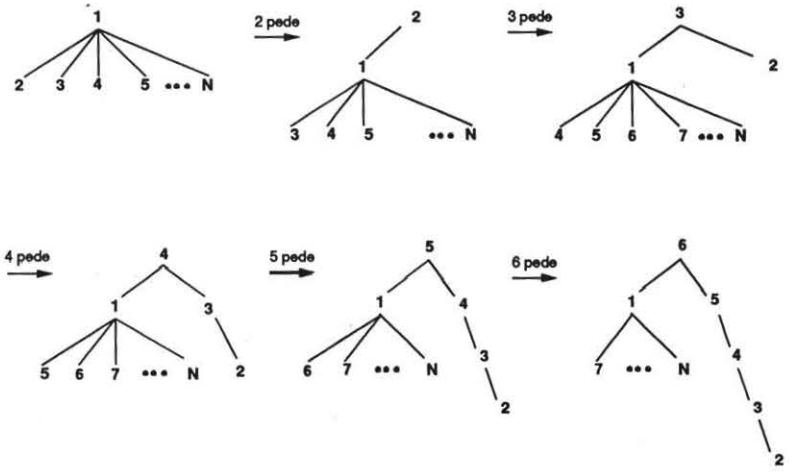

Figura 7: Seqüência de transições para gerar um pior caso

Nesta situação (sistema equilibrado e ocioso), se um nodo que já tenha utilizado solicitar o recurso novamente antes que todos os outros tenham feito acesso ao recurso, toda a cadeia será desfeita. Isso destaca uma característica importante do algoritmo que torna a formação de cadeias profundas um pouco rara que é o fato de algum elemento interno a uma cadeia destruí-la sempre que pedir o recurso; e quanto maior a cadeia maior a chance de algum de seus membros internos solicitar o recurso. A quebra de uma cadeia profunda leva, também, a um equilíbrio na árvore, visto que os nodos passam a possuir um número elevado de filhos. Então, a árvore tende a ser equilibrada e não muito profunda.

5.3 A Ordem de Complexidade

Nesta seção é feita uma análise da complexidade do algoritmo proposto, considerando que a operação relevante é a troca de mensagens. Um nodo, no melhor caso, não necessita trocar nenhuma mensagem para ter acesso ao recurso, basta o token estar em seu poder quando sua demanda ocorrer. O pior caso ocorre quando a árvore tem o formato de uma cadeia única, o token está em um dos extremos e o pedido trafega de um extremo ao outro. Neste caso, N mensagens são necessárias.

Apesar das inúmeras configurações possíveis de uma árvore de execução, nada impede que um sistema tenha uma árvore com uma profundidade média. Isso sugere que a ordem de complexidade do algoritmo neste sistema seja esta profundidade média, e como se trata de uma árvore, sugere que seja $O(\log N)$.

Porém, em determinadas cargas, o número de raízes é substancial, reduzindo a profundidade média das cadeias. Com base nisto e no que foi visto na seção anterior, pode-se analisar a ordem de complexidade do algoritmó conforme a carga do sistema. Em um sistema saturado todos os nodos serão raízes, o que faz o algoritmo ter complexidade constante. Mas a medida que há uma diminuição da carga a profundidade das cadeias começa a aumentar até alcançar um sistema ocioso onde há apenas uma raiz e na média a árvore é equilibrada e não muito profunda, o que dá ao algoritmo uma ordem de complexidade igual a $O(\log N)$.

Nosso algoritmo tem o inconveniente do tamanho do token crescer de acordo com o número de nodos do sistema ($N * \log N$ bits). Na maioria dos algoritmos propostos na literatura o tamanho da mensagem é constante. Em termos práticos, porém, esta característica não prejudica a utilização do algoritmo.

6 Resultados Numéricos

6.1 O Modelo de Comparação

O objetivo do Modelo de Comparação é nortear as simulações e a coleta de dados de modo a obter resultados claros e relevantes para a comparação entre algoritmos de exclusão mútua distribuída.

Pode-se dividir as variáveis do problema em variáveis do ambiente e variáveis dependentes da carga. As variáveis do ambiente são o tempo médio de repouso dos nodos na rede, o tempo médio de utilização do recurso e o tempo médio de transmissão de uma mensagem num canal da rede, $\overline{R}$, $\overline{U}$ e $\overline{d}$, respectivamente, além do número de nodos **N** da rede. O tempo de repouso **R**, neste modelo, poderá variar de zero até infinito, ou seja, um dado nodo pode nunca requerer o recurso. Já os tempos de transmissão **d** e utilização **U** devem ser garantidamente finitos; para assim garantir o tempo de espera para a utilização também finito, ou seja, não ocorra *starvation*.

Uma comparação de grandezas entre os valores $\overline{R}$ e $\overline{U}$ em um sistema fornece uma noção da carga em que o sistema se encontra. Se $\overline{R} \ll \overline{U}$, ao terminar sua utilização um nodo pede imediatamente o recurso, antes mesmo que o próximo nodo termine a sua utilização. Configura-se então uma sistema em saturação, que é caracterizado por uma fila média de nodos esperando com tamanho igual a $N-1$. Se $\overline{R} \gg \overline{U}$, ao pedir o recurso um nodo não tem nenhuma concorrência. Determina-se então um sistema em estado ocioso, que é caracterizado por uma fila média de nodos esperando com tamanho igual a 0. Dados estes dois limites, fixando-se o valor de $\overline{U}$ e variando $\overline{R}$ de valores suficientemente menores que $\overline{U}$ até valores suficientemente maiores que $\overline{U}$, o sistema varia de um estado de saturação até um estado ocioso, apresentando também valores de carga intermediários.

A quantificação da carga em sistemas como este pode ser dada pelo tamanho médio da fila de espera $\overline{f}$, que corresponde ao número de nodos em conflito por uma determinada utilização do recurso e que depende da relação $\frac{\overline{U}}{\overline{R}}$, do algoritmo usado no sistema e do número de nodos N da rede. Ou seja, definindo um algoritmo e estabelecendo $\overline{U}, \overline{R}$ e N, tem-se neste sistema uma fila de espera média $\overline{f}$, que representa a carga deste sistema. Com isso percebe-se que um algoritmo pode atingir uma carga específica (a máxima, por exemplo) em valores diferentes de $\frac{\overline{U}}{\overline{R}}$ se houver um número diferente de nodos na rede. Da mesma forma, algoritmos diferentes podem atingir a carga máxima em valores diferentes de $\frac{\overline{U}}{\overline{R}}$ para o mesmo número de nodos. Saber qual atingiu uma determinada carga mais rapidamente ($\overline{U}e\overline{R}$ com menor diferença), porém, não é um resultado tão interessante, já que diferentes algoritmos comportam-se de diferentes maneiras perante uma dada carga e atingí-la primeiro pode ser bom para alguns mas não significar nada para outros.

As variáveis que definem a performance de um algoritmo dependem todas da carga, assim deve-se sempre vincular a performance de um algoritmo a uma determinada carga. O número médio de mensagens trocadas por demanda de seção crítica $\overline{M}$, o tempo médio de espera para utilização do recurso $\overline{W}$[DHP85] e o atraso de sincronização médio $\overline{D}$ são variáveis dependentes de N, do algoritmo e da carga. Assim, para um dado N, pode-se calcular esses valores variando a carga e obtendo gráficos que servem tanto para analisar as performances dos algoritmos quanto compará-las.

$\overline{M}$ é o número de mensagens trocadas por demanda. Esse valor indica uma importante medida da performance do algoritmo, pois quanto maior $\overline{M}$, maior a pressão de mensagens sobre a rede. $\overline{M}$ porém não reflete a ordem na qual essas mensagens são transmitidas pelo algoritmo. Para isso temos $\overline{m}$, que corresponde ao fator que multiplicado por $\overline{d}$ resulta no tempo gasto para a transmissão das $\overline{M}$ mensagens necessárias para a utilização do recurso. O valor de $\overline{m}$ é no mínimo 2, no caso em que os nodos transmitem seus pedidos de uma só vez, e recebem as permissões necessárias também de uma só vez; e no máximo $\overline{M}$, quando as mensagens são trocadas em seqüência.

O Atraso de Sincronização médio $\overline{D}$ é o tempo médio que um nodo espera para utilizar o recurso sendo que não há ninguém o utilizando e é a sua vez de utilizá-lo. Seu valor é, então, dado pela porção de $\overline{md}$ que é executada quando o recurso está ocioso e está na vez do nodo utilizá-lo. Para quantificar esta porção definimos $\overline{\lambda}$, que então varia de $\frac{\overline{m}}{\overline{m}} = 1$ até $\frac{\mathbf{k}}{\overline{m}}$. É igual a 1 quando todas as $\overline{M}$ mensagens são transmitidas sem que ninguém esteja utilizando o recurso e o nodo será o próximo a usá-lo, ou seja, quando temos um sistema ocioso. E é igual a $\frac{\mathbf{k}}{\overline{m}}$ quando a porção de $\overline{m}$ transmitida fora dos tempos de utilização e troca de mensagens dos nodos mais prioritários for correspondente apenas ao envio da permissão para o nodo (que é o mais prioritário então) entrar na seção crítica (valor definido pela variável **k**, e por isso $\frac{\mathbf{k}}{\overline{m}}$). Sendo assim $\overline{D} = \overline{md}$ em sistemas ociosos e $\overline{D} = \overline{kd}$ em sistemas com cargas médias altas. Em sistemas com valores intermediários de carga, teremos $\overline{\lambda}$ com seus valores intermediários e $\overline{D}$

$= \overline{\lambda m d}$. Para comparar $\overline{D}$ é interessante que ele esteja normalizado em relação aos tempos, assim $\overline{D_n} = \frac{\overline{D}}{\overline{d}}$, ou $\overline{D_n} = \overline{\lambda m}$.

O tempo médio de espera de um nodo pela sua vez de utilizar o recurso $\overline{W}$ pode ser visto como o tempo para a utilização $\overline{U}$ do recurso pelos nodos mais prioritários, acrescido dos atrasos de sincronização para cada uma dessas utilizações e para a utilização do nodo em questão. Assim, $\overline{W}$ seria:

$$\overline{W} = \overline{f}\overline{U} + (\overline{f}+1)\overline{D}.$$

Os valores de $\overline{W}$ variam com a carga indo de $\overline{W} = \overline{D} = \overline{md}$ em sistemas ociosos até $\overline{W} = (N-1)\overline{U} + N\overline{kd}$ em sistemas saturados. Como em $\overline{D}$, para comparar $\overline{W}$ é interessante que ele esteja normalizado em relação aos tempos. Fazendo $\overline{U} = \overline{d}$, temos

$$\overline{W_n} = \frac{\overline{W}}{\overline{d}} = \overline{f} + (\overline{f}+1)\overline{D_n}.$$

Apesar de $\overline{m}$ ser a variável mais importante para se avaliar a performance de um algoritmo de exclusão mútua distribuída, os valores de $\overline{M}$, $\overline{W_n}$ e $\overline{D_n}$ trazem uma grande quantidade de informações (inclusive $\overline{m}$, apesar de indiretamente) sobre o comportamento geral dos algoritmos nas variadas cargas. $\overline{M}$, $\overline{W_n}$ e $\overline{D_n}$ são então variáveis adequadas para analisar e comparar tais algoritmos.

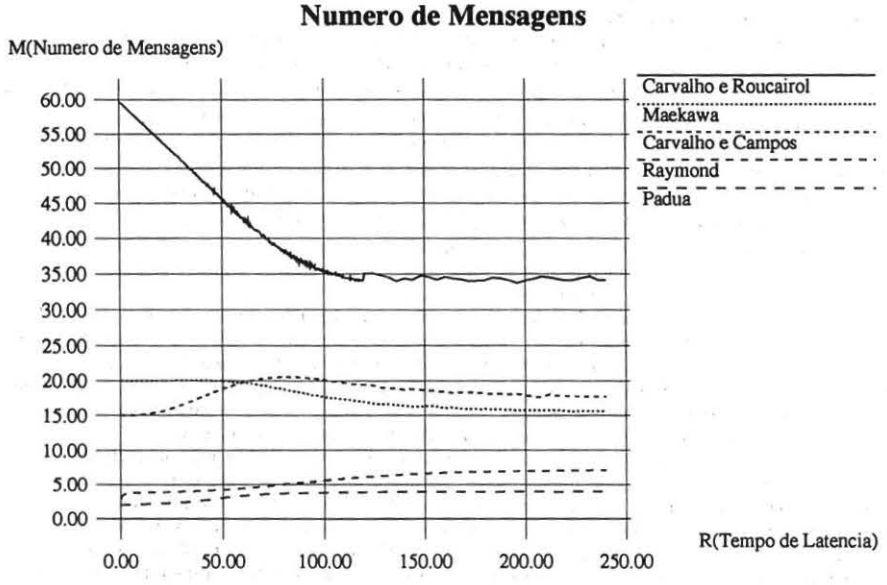

Figura 8: Gráficos $\overline{M}$ X $\overline{R}$

6.2 Os Resultados das Simulações

O ambiente de simulação utilizado para realizar as medidas aqui apresentadas não implementa completamente o modelo de simulação proposto. A principal falta é a medida da fila de espera média($\overline{f}$).

Os algoritmos usados nas simulações, além do aqui apresentado, foram Carvalho & Roucairol [CR83], Maekawa [Mae85], Carvalho & Campos [CC90] e Raymond [Ray89]. O primeiro possui ordem de complexidade $O(N)$ para o número de mensagens trocadas por demanda e foi escolhido por ser o primeiro

a introduzir idéias como sonegação e permissão implícita (não é sempre necessário mandar os pedidos a todos os sítios), o que mudou a maneira de ver o problema. O segundo tem ordem de complexidade para o número de mensagens trocadas por demanda igual a $O(\sqrt{N})$, pois usa um plano projetivo finito para reduzir o número de sítios para onde cada nodo deve mandar seus pedidos. O terceiro também é $O(\sqrt{N})$ pois também usa plano projetivo finito, além de algumas idéias do *drinking philosophers* de Chandy & Misra [CM84] que o torna mais eficiente em cargas elevadas. O último tem uma complexidade de ordem $O(\log N)$ pois estrutura os nodos numa árvore fixa.

Figura 9: Pádua: $\overline{W}$ X $\overline{R}$ e $\overline{D}$ X $\overline{R}$

O gráfico da figura 8 contém os valores de $\overline{M}$ dos 5 algoritmos para um sistema com 31 nodos, $\overline{U} = 1.0$ e $\overline{R}$ variando de 1.0 até 250, que não chega a alcançar cargas extremas. Mas mesmo assim pode-se observar claramente os valores de $\overline{M}$ indo para seus limites nestas cargas, além de fornecer características interessantes de seus comportamentos nas diversas outras cargas. O gráfico mostra de uma maneira bem interessante como as faixas em que se encontram as curvas dos algoritmos se separam de acordo com sua ordem de complexidade. A faixa dos $O(N)$ vem acima, no meio está a faixa dos $O(\sqrt{N})$ e mais abaixo, a faixa dos $O(\log N)$. Pelo gráfico, o algoritmo apresentado neste trabalho consegue os menores valores para $\overline{M}$ em todas as cargas.

O gráfico da figura 9 contém os valores de $\overline{W_n}$ e $\overline{D_n}$ do nosso algoritmo para um sistema com 31 nodos, $\overline{U} = 1.0$ e $\overline{R}$ variando de 1 até 250. Ele confirma o que o modelo havia previsto sobre o atraso de sincronização se aproximar do tempo de espera em sistemas ociosos. Além disso, comprova a fórmula de $\overline{W_n}$ em saturação como sendo $(N-1)+N\overline{k}$, pois $\overline{W_n} = 61$ onde $N = 31$ e o $\overline{k}$ deste algoritmo, como já vimos, é sempre 1. Mostra que mesmo fora da saturação, o $\overline{D_n}$ continua mínimo durante as cargas altas ($= \overline{k}$) e então vai aumentando lentamente ($\overline{\lambda}$ crescendo) em direção ao seu valor máximo ($\overline{m}$) que será igual ao $\overline{W_n}$ quando em ociosidade.

O gráfico da figura 10 contém os valores de $\overline{W_n}$ dos 5 algoritmos para um sistema com 31 nodos, $\overline{U} = 1.0$ e $\overline{R}$ variando de 1 até 250. Aqui, como no $\overline{M}$ X $\overline{R}$, pode-se observar as curvas saindo de seus valores previstos em saturação e indo para seus valores previstos em ociosidade (que se aproximam dos atrasos de sincronização). É interessante notar o elevado tempo de espera do Raymond, que se justifica pelo seu alto $\overline{m}$, já que sua árvore é fixa e para atravessá-la as mensagens têm que percorrer os nodos vizinhos um de cada vez.

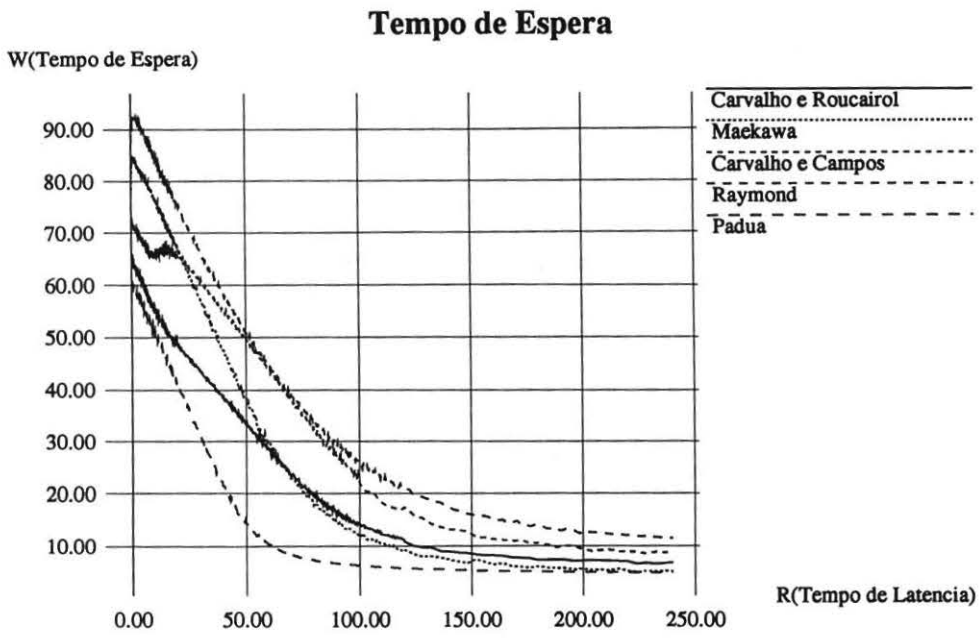

Figura 10: Gráficos $\overline{W}$ X $\overline{R}$

7 Conclusão

Esse trabalho utilizou um ambiente de simulação [CMC94] bastante poderoso e que provou isto por ter possibilitado aos autores implementarem vários algoritmos e compreenderem melhor as variáveis do problema e suas relações. Compreensão esta que culminou no desenvolvimento do algoritmo e do modelo de comparação apresentados.

Os testes aqui realizados mostram que o algoritmo aqui proposto apresenta uma boa performance em relação às variáveis consideradas relevantes. Tal fato estimula o prosseguimento dos trabalhos, visto que o algoritmo ainda se encontra em fase de desenvolvimento. O principal complemento a ser anexado à esta versão do algoritmo é um mecanismo de tolerância a falhas dos nodos.

Cabe ressaltar que o algoritmo aqui descrito só foi implementado num ambiente de simulação. Tenciona-se implementá-lo, também, em um ambiente NOW (Network Of Workstations) e realizar comparações e medidas sobre ele. Esta implementação possibilitará também determinar a praticidade deste algoritmo em um ambiente distribuído real.

Referências

[Lam78] Lamport, L. *Time, clocks, and ordering of events in a distributed system.* Commun. ACM 21,7 (July, 1978), 558-565.

[Ray89] Raymond, K. *A Tree-Based Algorithm for Distributed Mutual Exclusion.* Commun. ACM 1,7 (February, 1989), 61-77.

[SK86] Suzuki, I., and Kasami, T. *A distributed mutual exclusion algorithm* ACM Trans. Comput. Syst. 3,4 (November, 1985), 344-349.

[CM84] Chandy, K.M., and Misra J. *The drinking philosophers.* ACM Trans. Program. Lang. Syst. 6,4 (October, 1984), 632-646.

[DJK65] Dijkstra, E.W. *Co-operating sequential processes.* in Programming Languages, Genuys, F. (ed.), Academic Press, London, 1965

[DJK75] Dijkstra, E.W. *Guarded commands, nondeterminacy and formal derivation of programs.* Commun. ACM 18,8 (August, 1975), 453-457.

[RA81] Ricart, G., and Agrawala, A.K. *An Optimal algorithm for mutual exclusion in computer networks.* Commun. ACM 24,1 (January, 1981), 9-17.

[CR83] Carvalho, O.S.F., and Roucairol,G. *On mutual exclusion in computer networks.* Commun. ACM 26,2 (February, 1983), 146-147.

[CC90] Carvalho, Osvaldo S.F. and Campos, Sérgio V.A. *A $0..\sqrt{n}$ distributed mutual exclusion algorithm.* Personal Notes.

[Mae85] Maekawa, Mamoru. *A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems.* ACM trans. Comp. Syst. 3,2 (May, 1985), 145-159.

[AAH89] Bernabéu-Aubán, J. M., and Ahamad, M. *Applying a path compression technique to obtain an efficient distributed mutual exclusion algorithm.* In Proceedings of Third International Workshop on Distributed Algorithms (September, 1989), 33-44

[DHP85] Dupuis, Alan and Hebuterne, Gérard and Pitie, Jean-Marc. *A Comparison of Two Mutual Exclusion Algorithms for Computer Noteworks.* Note Technique CNET/LAA/SLÇ 1985.

[CMC93] Couto, K.O., Mendes, M.A.S., e Carvalho, O.S.F. *Uma Comparação Entre Dois Algoritmos de Exclusão Mútua para Redes de Computadores.* ANAIS of V SBAC-PAD, Vol I, 358-367.

[CMC94] Couto, K.O., Mendes, M.A.S., e Carvalho, O.S.F. *Ambiente de Desenvolvimento e Análise de Algoritmos de Exclusão Mútua para Sistemas Distribuídos.* ANAIS of VI SBAC-PAD, 123-135.