Um algoritmo para exclusão mútua em sistemas distribuídos com troca de $O(\sqrt{n})$ mensagens de 2 bits

Sérgio V. A. Campos* Osvaldo S. F. Carvalho*

Out 1988

Sumário

Este trabalho apresenta um algoritmo para exclusão mútua em sistemas distribuídos que troca um número de mensagens por invocação de seção crítica que varia entre 0 e $4\sqrt{n}$, onde n é o número de nodos participantes do algoritmo. Conflitos são resolvidos utilizando-se um grafo acíclico de prioridades conforme a proposta de Chandy e Misra para resolução do problema dos *Filósofos Famintos* (The Dining Philosophers). A não utilização de Timestamps permite reduzir a dois bits o tamanho das mensagens trocadas.

Abstract

This paper presents an algorithm to achieve mutual exclusion in distributed systems that exchanges a number of messages per critical section invocation that varies from 0 to $4\sqrt{n}$, where n is the number of participant nodes. Conflicts are solved using an acyclic graph of priorities, as proposed by Chandy and Misra to the resolution of the problem of *The Dining Philosophers*. The absence of timestamps allows the messages to be coded in 2 bits.

*Esta pesquisa foi parcialmente financiada com recursos do CNPq e da SID Informática - Projeto ESTRA

1 Introdução

O estudo de algoritmos eficientes para exclusão mútua entre processos de um sistema distribuído é de grande importância prática, visto que esta é uma ferramenta essencial para resolução de todo problema de sincronização. Para que um algoritmo seja aceitável é preciso que ele satisfaça três propriedades fundamentais: a equanimidade (fairness), a ausência de impasses (deadlocks) e a simetria. Por algoritmo simétrico entende-se aqui que cada nodo deve ter as mesmas responsabilidades que qualquer outro na execução do algoritmo, e deve também ter as mesmas possibilidades de entrada na região crítica.

Para se medir a eficiência de um algoritmo, tem-se tomado como parâmetro mais relevante o número de mensagens trocadas por invocação de seção crítica [2,3,4]. Ricart e Agrawala [4] propuseram um algoritmo que necessita de $2(n-1)$ mensagens para se conseguir a exclusão mútua, onde n é o número de nodos participantes. A concordância de todos os nodos é exigida para que se possa ter o acesso exclusivo ao recurso disputado. Conflitos são resolvidos com base em "timestamps" e relógios lógicos [10], e é garantido que as requisições serão atendidas respeitando-se a ordem FCFS dos relógios lógicos amostrados nos instantes das requisições. Carvalho e Roucairol [6] propuseram uma modificação deste algoritmo que utiliza entre 0 e $2(n-1)$ mensagens por invocação de seção crítica. A resolução de conflitos também é baseada em relógios lógicos e "timestamps", mas a ordem de atendimento das requisições não é necessariamente FCFS. O limite inferior de 0 mensagens trocadas para se conseguir a exclusão mútua é conseguido aplicando-se a seguinte idéia: se um nodo i consegue a autorização de um nodo j para entrar na seção crítica, esta autorização permanece válida até que j a requisite novamente.

Maekawa [2] propôs um algoritmo que utiliza $c\sqrt{n}$ mensagens, onde $3 <= c <= 5$. Neste algoritmo associa-se a cada nodo um conjunto S_i de outros nodos aos quais serão enviados pedidos de exclusão mútua. Cada nodo i possui, em consequência, um conjunto R_i de nodos que lhe pedem autorização para entrar na seção crítica, e age como árbitro destes nodos, garantindo que no máximo um dentre eles obtenha esta autorização a cada vez. A garantia de exclusão mútua global é conseguida construindo-se os conjuntos S_i de tal maneira que qualquer par de nodos i e j possua pelo menos um árbitro comum (i.e. $S_i \cap S_j \neq \emptyset$). Também aqui utilizam-se relógios lógicos para a resolução de conflitos, mas não se garante o respeito à ordem FCFS para o atendimento das requisições. A grande economia de mensagens deste algoritmo resulta do uso de um processo criterioso para a escolha dos conjuntos S_i, baseado em propriedades de *Planos Finitos Projetivos* [3,11]

Uma alternativa para a resolução de conflitos em sistemas distribuídos foi proposta por Chandy e Misra [1] para a solução dos problemas dos Filósofos Famintos (The Dining Philosophers) e dos Filósofos Sedentos (The Drinking Philosophers), que são generalizações do famoso problema proposto por Dijkstra [8]. A profundidade em um grafo acíclico[1] é utilizada como propriedade de distinção entre requisições conflitantes. A aciclicidade deste grafo impede a

[1] A profundidade de um nodo em um grafo acíclico é definida como sendo a maior distância entre o nodo e qualquer dos nodos sem predecessores.

formação de bloqueios, e o grafo é modificado de forma a garantir que todo nodo ascenda à profundidade zero após um número finito de conflitos perdidos, o que garante a equanimidade do algoritmo. Poucos tipos distintos de mensagens são utilizados, o que permite limitar o tamanho das mensagens trocadas a poucos bits. Convém aqui lembrar que, ao menos em princípio, algoritmos que utilizam timestamps não possuem limite para o tamanho das mensagens trocadas.

A solução dada por Chandy e Misra para o problema dos filósofos famintos pode ser diretamente adaptada para se conseguir a exclusão mútua, sendo que o número de mensagens trocadas seria entre 0 e $2(n-1)$, tendo o algoritmo, neste caso, um comportamento próximo ao proposto por Carvalho e Roucairol [6].

O algoritmo aqui proposto resulta da fusão das idéias de Maekawa e de Chandy e Misra. Planos finitos projetivos são utilizados para organização dos nodos nos conjuntos S_i e R_i, a profundidade num grafo acíclico de prioridades é utilizada para a resolução de conflitos, e autorizações permanecem válidas até que sejam reclamadas. Com isto foi possível conseguir um algoritmo distribuído de exclusão mútua que é livre de bloqueios, equânime, simétrico, utiliza mensagens de somente dois bits e é eficiente pois necessita trocar entre 0 e $4\sqrt{n}$ mensagens por invocação de região crítica.

A organização deste trabalho é a seguinte: A seção 1 apresenta uma introdução ao algoritmo proposto. Na seção 2 segue-se uma explicação sobre os *Planos Finitos Projetivos* e sua aplicação para se conseguir a exclusão mútua. Na seção 3 é apresentado o problema dos *Filósofos Famintos*. A seção 4 descreve o algoritmo e é dividida em três subseções, 4.1 que expõe a idéia básica do algoritmo, 4.2 aonde é descrito formalmente o algoritmo e 4.3, que descreve a implementação da lista de prioridades usada pelo algoritmo. É apresentada a prova do algoritmo na seção 5. Finalmente, a seção 6 discute a complexidade e a seção 7 é a conclusão do trabalho. A seção 8 apresenta as referências bibliográficas.

2 Planos Finitos Projetivos e o Esquema de Comunicações

2.1 Introdução

Diversos algoritmos distribuídos de exclusão mútua utilizam trocas de "autorizações" entre nodos para entrada na seção crítica. Nesses algoritmos pode-se identificar para cada nodo i um conjunto S_i de nodos aos quais ele pede diretamente esta autorização, assim como um conjunto R_i de nodos que lhe pedem autorização.

Por exemplo, nos algoritmos [1,4,6] cada nodo pede diretamente autorização a todos os outros. É importante notar que, apesar da autorização de todos os outros nodos ser necessária para que um nodo possa entrar em sua seção crítica, isto muitas vezes pode ser feito de maneira indireta, com alguns nodos agindo como "representantes" de outros. Podemos por exemplo ter um algoritmo com um coordenador central que age como representante de todos os outros

ao enviar uma autorização para qualquer de seus coordenados.

A autorização indireta é um instrumento poderoso de economia de tráfego de mensagens, mas pode induzir assimetrias e demoras no tempo de resposta do algoritmo, devido à necessidade de se percorrer rotas muito longas para se obter informações sobre um nodo do qual só se tem acesso através do seu representante, ou do representante de seu representante, etc. Sugere-se a determinação dos conjuntos S_i por meio de *planos finitos projetivos*, o que apresenta as seguintes vantagens:

- o número de nodos aos quais um nodo pede diretamente autorizações cresce com a raiz do número total de nodos no sistema, o que permite uma grande economia de mensagens;
- todo nodo pede autorizações ao mesmo número de nodos, o que permite a construção de algoritmos simétricos;
- um nodo i pode obter a autorização de qualquer nodo j com a interposição de no máximo um representante.

Formalmente, os conjuntos S_i obtidos por meio de planos finitos projetivos apresentam as seguintes propriedades:

$$\forall i \forall j \ (S_i \cap S_j) \neq \emptyset, \ |S_i \cap S_j| = 1, \ |S_i| = |S_j|$$

2.2 Planos Finitos Projetivos

Passamos agora a uma curta exposição sobre planos finitos projetivos (daqui por diante chamados de PFPs), seguindo os passos de Lakshman e Agrawala [3].

Um plano finito projetivo consiste de um conjunto finito de pontos e linhas, onde linhas são conjuntos de pontos, que satisfazem aos seguintes postulados:

P 1 *Dois pontos distintos estão em uma e somente uma linha comum.*

P 2 *Duas linhas distintas passam por um e somente um ponto comum.*

P 3 *Existem quatro pontos distintos, sendo que três deles não estão na mesma linha.*

O postulado 3 é necessário para eliminar certos PFP degenerados como por exemplo um conjunto de pontos e uma única linha (Para esta aplicação, porém, planos finitos projetivos como um triângulo ou uma linha com somente dois pontos são válidos).

Os teoremas a seguir estabelecem propriedades dos PFP que serão úteis à solução do problema aqui discutido.

Teorema 1 *Em um plano finito projetivo, cada ponto pertence ao mesmo número de linhas, e cada linha passa pelo mesmo número de pontos.*

Teorema 2 *Em um plano finito projetivo, o número de linhas que passam através de cada ponto é igual ao número de pontos em cada linha.*

Teorema 3 *Um plano finito projetivo com $m+1$ pontos em cada linha e $m+1$ linhas passando por cada ponto tem m^2+m+1 pontos e m^2+m+1 linhas.*

O número m no teorema 3 é chamado de *ordem* do plano finito projetivo.

Teorema 4 *Se $m = p^k$ para p primo e k um inteiro positivo, então existe um plano finito projetivo de ordem m.*

Como exemplo apresenta-se um plano finito projetivo de ordem 2. É constituido dos pontos $\{1,2,3,4,5,6,7\}$ e das linhas:

$$L_1:\{1,2,4\}\ L_2:\{2,6,7\}\ L_3:\{3,4,6\}\ L_4:\{4,5,7\}\ L_5:\{5,2,3\}\ L_6:\{6,5,1\}\ L_7:\{7,3,1\}$$

2.3 Aplicações dos PFPs ao Problema da Exclusão Mútua

Para determinar os conjuntos S_i e R_i por meio de planos finitos projetivos, toma-se cada nodo por um ponto e constrói-se um PFP com todos os nodos do sistema. Se não existir um PFP com este número de pontos (veja o teorema 4), acrescentam-se nodos fictícios[2] a fim de que se consiga um número de nodos para o qual exista um PFP. O conjunto S_i é definido pela linha L_i. O conjunto R_i tem como elementos os nodos j tais que $i \in S_j$. O nodo i comunica-se com os nodos pertencentes a S_i e R_i. Pode-se ainda conseguir economias na comunicação construindo o PFP de maneira a termos sempre $i \in S_i$ (um PFP pode ser construido de maneira a termos sempre L_i contendo o ponto i, como no exemplo acima [3]) .

O postulado 2 e o teorema 1 garantem que o graíο de comunicações é conexo e simétrico. Os teoremas 2 e 3 garantem que $|S_i| = |S_j| \approx \sqrt{n}$, onde n é o número de nodos do sistema.

3 O Problema dos *Filósofos Famintos*

No problema dos Filósofos Famintos (The Dining Philosophers), proposto e resolvido por Chandy e Misra, cada nodo de um sistema distribuído é representado por um filósofo que passa a sua vida pensando e comendo. Os filósofos dividem com cada um dos seus vizinhos um "garfo". Cada garfo é disputado por exatamente dois filósofos, definindo uma aresta de um graíο G de conflitos que tem como vértices todos os nodos do sistema. Para comer ele precisa de todos os garfos que divide com seus vizinhos. Diz-se que um filósofo está faminto durante o período em que pretende comer mas ainda não conseguiu os garfos de que necessita.

[2] Pode-se mostrar [3] que mesmo com o acréscimo de nodos fictícios o número de vizinhos de cada nodo permanece $O(\sqrt{n})$.

Trata-se de uma generalização do famoso problema proposto por Dijkstra [8], onde o grafo de conflitos seria sempre circular.

Conflitos são resolvidos por meio de um grafo dirigido H, que possui os mesmos vértices e arestas do grafo G. Uma aresta de H é dirigida de um nodo p para um nodo q se p tem precedência sobre q, isto é todo conflito entre p e q seria resolvido em favor de p. O algoritmo mantém uma representação distribuída do grafo H, que é modificado dinamicamente. As modificações são feitas de forma a preservar a aciclicidade de H, o que faz com que a profundidade de um nodo em H possa sempre ser utilizada para se decidir sobre a precendência de um ou de outro nodo na ocorrência de conflitos, e fazer com que todo nodo em conflito ascenda eventualmente ao topo de H, o que garante a equanimidade do algoritmo.

O grafo H é inicialmente acíclico, e sua aciclicidade é preservada num ambiente distribuído pela aplicação da seguinte regra:

Regra da Aciclidade: Todas as arestas ligadas a um nodo p podem ser simultaneamente dirigidas para p.

Esta transformação preserva a aciclicidade de H, visto que, como todas as arestas ligadas a p estarão dirigidas para ele, nenhum novo ciclo contendo p poderá ser formado.

O algoritmo funciona da seguinte maneira. É pressuposto um meio de comunicação que não perde e nem deturpa mensagens, entregando-as sempre na mesma ordem em que forem enviadas. A cada garfo associa-se uma mensagem (um "token") que circula entre os dois filósofos que o disputam. Um garfo está limpo ou sujo. Ao ser utilizado para comer, garfo fica sujo e permanece sujo até ser enviado; o envio constitui a única forma possível de se limpar um garfo.

Requisições são comunicadas por meio de um "request token" que também circula entre dois filósofos potencialmente conflitantes; somente o filósofo de posse deste "request token" pode enviá-lo. Um filósofo que está comendo ignora (adia a resposta) todas as requisições que recebe; um filósofo pensando atende a todas as requisições que recebe; um filósofo faminto adia a resposta para todas as requisições provenientes de nodos com os quais ele compartilha um garfo do qual ele está de posse e que está limpo, e atende a todas as requisições provenientes de nodos que disputam com ele garfos que estão sujos. Quanto ao desempenho, é fácil ver que se o número máximo de filósofos com quem um filósofo qualquer do sistema disputa garfos é d, para que um filósofo consiga comer são enviadas no máximo $2d$ mensagens, sendo d requisições e d garfos. Vale observar que só são usadas mensagens de dois tipos distintos, podendo portanto ser codificadas com um único bit. Detalhes do algoritmo podem ser vistos em [1].

A direção de uma aresta que une dois visinhos p e q no grafo H é representada da seguinte forma: p precede q em H (isto é, p tem precedência sobre q) se e somente se:

1. p possui o garfo g_{pq} e ele está limpo, ou se

2. q possui o garfo g_{pq} e ele está sujo, ou se

3. o garfo g_{pq} está em trânsito de q para p.

É importante observar que a direção de uma aresta só é modificada quando um filósofo começa a comer, e ainda que quando um filósofo come, ele suja todos os seus garfos simultaneamente, o que é uma implementação da regra da aciclicidade acima. É fácil ver também que nenhum filósofo faminto pode perder dois conflitos para um mesmo vizinho durante um mesmo período de fome, o que garante que todo filósofo comerá após um período finito de fome.

Para se resolver o problema da exclusão mútua pelo algoritmo dos Filósofos Famintos, associa-se a todo par de nodos i e j um garfo g_{ij} que será disputado por eles. Um nodo i poderá ter acesso à sua seção crítica quando estiver de posse de todos os $(n-1)$ garfos g_{ij}. Como um garfo só pode estar com no máximo um dentre os seus dois disputantes (um garfo pode também estar "em trânsito"), este esquema garante a exclusão mútua. Em outras palavras, o problema dos Filósofos Famintos se reduz ao problema da exclusão mútua quando o grafo de conflitos G é completo. Cada filósofo está em conflito com todos os $(n-1)$ outros filósofos, o que faz com que o algoritmo troque entre 0 e $2(n-1)$ mensagens por invocação de seção crítica, tendo um comportamento bastante próximo ao de Carvalho e Rouicarol [6].

4 O Algoritmo

4.1 A Idéia do Algoritmo

O algoritmo aqui proposto resulta da fusão das idéias de Maekawa e de Chandy e Misra. Mantemos a idéia dos garfos g_{ij} disputados entre todo par de nodos i e j, mas utilizando um esquema de comunicações gerado por um plano finito projetivo.

Cada nodo desempenha tanto o papel de cliente como de árbitro. Para simplificar a descrição do algoritmo, utilizamos dois processos — um cliente e um árbitro — por nodo para a execução destas tarefas.

Numa fase preliminar, forma-se um plano finito projetivo com os nodos do sistema, acrescentando-se os nodos fictícios eventualmente necessários. Cada linha L_i do PFP define o conjunto S_i de árbitros a quem o cliente i deve pedir autorização. Por motivo de economia, o cliente i sempre pede autorização para o árbitro i.

Mensagens só são trocadas entre um cliente i e um árbitro j se $j \in S_i$; um cliente nunca se comunica com outro cliente, nem um árbitro se comunica com outro árbitro. As mensagens que circulam entre um cliente c e um árbitro a são um "token" $FORK_a$ que, como veremos, representa um conjunto de garfos que podem estar (todos) sujos ou (todos) limpos, e um "token" de requisição.

Para entrar na seção crítica, um nodo i precisa de todos os garfos g_{ij}. Entretanto, tanto a representação da presença dos garfos como a sua transmissão são feitos de forma indireta. Consideremos dois nodos i e j quaisquer. Pelo postulado 1, existe sempre um único árbitro a tal que $a \in S_i$ e $a \in S_j$. Nós dizemos que o garfo g_{ij} está com o cliente i se e somente se i

está de posse do token $FORK_{ai}$. O mesmo vale para o cliente j. Se o árbitro a está de posse dos tokens $FORK_{ai}$ e $FORK_{aj}$, nós dizemos que o garfo g_{ij} está em posse do árbitro a.

Quando um árbitro a transmite o token $FORK_{ai}$ ao seu cliente i, pela definição acima ele estará transmitindo todos os garfos g_{ik} onde $k \in R_a$. Um árbitro deve então estar sempre de posse de todos os seus tokens com exceção possível de um único, pois, caso contrário, poderia ocorrer que tanto um cliente i quanto um cliente j pensassem estar de posse do garfo g_{ij}. Desta forma, pode-se dizer que cada árbitro administra um recurso de uso mutuamente exclusivo entre seus clientes.

Aqui também garfos podem estar sujos ou limpos. Ao comer, um cliente suja todos os seus garfos, que entretanto só são limpos *após* serem transmitidos a um árbitro. Como veremos, ao transmitir um $FORK$, um cliente deve informar ao árbitro destinatário se os garfos correspondentes estavam limpos ou sujos no momento da transmissão.

Para representar o grafo H de precedências, é necessário ainda que todo árbitro mantenha uma lista de prioridades que contém todos os seus clientes e que só pode ser alterada pela operação $GiveLeastPriorityTo(c : ClientId)$, que move um cliente designado para a posição de última prioridade.

A posição e o estado dos garfos juntamente com as listas de prioridades dos árbitros definem as relações de precedência em H: Sejam i e j dois clientes quaisquer e a o seu árbitro. O cliente i precede o cliente j em H se e somente se

1. j possui o garfo g_{ij}, que está sujo, ou se
2. o token $FORK_{aj}$ está sendo transmitido de j para a, e está sujo, ou ainda se
3. Nenhuma das condições acima se aplica nem ao nodo i e nem ao nodo j e $Priority_a(i) > Priority_a(j)$.

É interessante observar que por estas regras, nem sempre um árbitro consegue determinar a relação de precedência entre um cliente i para o qual ele (o árbitro) enviou um FORK e um outro cliente qualquer, posto que somente i sabe se seus garfos já estão sujos ou não.

Em linhas gerais o algoritmo funciona assim:

Quando fica com fome, um nodo i requisita todos os garfos dos árbitros seus vizinhos. Se o árbitro a estiver com todos os seus garfos (significando que é dono do seu próprio recurso), envia o garfo para i. Se a não estiver com todos os seus garfos, isto significa que o seu recurso foi entregue anteriormente para um nodo j também arbitrado por a. Neste caso, a requisitará o garfo de j para entregá-lo a i. Quando j terminar de comer, entregará o garfo a a, e ele será enviado a i.

Um problema ocorre porém, se a aresta ij do grafo H estiver direcionada de i para j, significando que, em H, i é mais prioritário que j. Neste caso, pode existir um caminho em H que leve de j a i (veja teorema 5) o que implica que j pode estar indiretamente esperando que i termine de comer para conseguir todos os garfos. Acontece que, se j entregar o garfo a a somente após comer, haverá um ciclo em H, com i esperando por j e vice-versa.

Neste caso, porém, como i é mais prioritário que j, j deve entregar o garfo para o mesmo que não tenha comido. Foi criado, desta forma, uma *requisição forte* (StrongRequest). Um árbitro envia uma requisição forte a um nodo quando há um nodo com maior prioridade desejando o garfo. Um cliente, ao receber uma requisição forte, deve entregar o garfo mesmo que não tenha comido.

Quando um cliente entrega o garfo ao árbitro, deve informar se conseguiu ou não comer. Isto é necessário para atualizar-se a lista de prioridades associada ao árbitro. Se o garfo foi enviado em resposta a uma requisição forte, a prioridade do nodo não deve necessariamente ser diminuida, pois ele pode não ter conseguido comer. Assim, quando um cliente envia um garfo a um árbitro, ele o envia sujo ou limpo. Se o garfo estiver sujo, o cliente conseguiu comer e sua prioridade deve ser diminuida. Se o garfo estiver limpo, o cliente não conseguiu comer e sua prioridade não deve ser alterada. O recebimento de um garfo limpo por um árbitro significa também o recebimento de uma requisição do garfo, pois o cliente ainda está com fome.

4.2 Descrição Formal

Existem três tipos de mensagens que podem ser enviadas por um árbitro a um cliente que são: *FORK*, *Request* e *StrongRequest*. De um cliente a um árbitro existem também três tipos de mensagens possíveis: *Request*, *CleanFORK* e *DirtyFORK*, sendo que quando um cliente envia um *CleanFORK* a um árbitro, é indicado também o envio de um *Request*. A fim de diminuir o número de mensagens, codifica-se-as em quatro tipos, *CleanFORK*, *DirtyFORK*, *Request* e *StrongRequest*. Desta forma, quando um árbitro envia uma mensagem *FORK* a um cliente, ele envia um *CleanFORK*. Não há diferenças entre o *Request* enviado pelo cliente e o enviado pelo árbitro. Sobrepondo-se as mensagens desta maneira consegue-se um número total de mensagens de apenas quatro, ao invés de seis que existiriam se isto não fosse feito. A sobreposição não gera ambiguidades porque sempre que uma mensagem tiver dois significados diferentes (como por exemplo o recebimento de um *CleanFORK* por um árbitro ou por um cliente), o contexto permite a diferenciação.

São usadas as seguintes variáveis booleanas no algoritmo:

$FORK_u(f)$:	O filósofo u tem a posse do garfo f.
$req_u(f)$:	O filósofo u tem a ficha de requisição para o garfo f.
$strongreq_u(f)$:	O filósofo u tem a ficha de requisição para o garfo f.
$dirty_u(f)$:	O garfo f está com o filósofo u e está sujo.
$Thinking_u$:	O filósofo u está *Pensando*.
$Hungry_u$:	O filósofo u está *Faminto*.
$Eating_u$:	O filósofo u está *Comendo*.

A variável $OWNER_u$ indica qual é o filósofo que está com o recurso do árbitro u, assumindo o valor NIL quando o recurso está com o próprio árbitro.

São usados também a funções *Priority* e *GiveLeastPriorityTo*, esta última movendo o filósofo que lhe for passado como parâmetro para a posição de menor prioridade na lista de prioridades associada ao árbitro. É usada também a função *HighestPriorityRequest* que retorna, dentre todos os clientes que desejam o $FORK$ de um árbitro, aquele que tem o maior valor de *Priority*.

Algoritmo do Cliente

R 1 A qualquer instante o filósofo pode ficar com fome:

$Thinking \rightarrow$
$Hungry := TRUE$

R 2 Quando estiver com todos os garfos, começa a comer:

$Hungry, \forall i\ FORK(i) = TRUE \rightarrow$
$Eating := TRUE$
$\forall k \in S \rightarrow dirty(k) := TRUE$

R 3 A qualquer instante o filósofo pode parar de comer

$Eating \rightarrow$
$Thinking := TRUE$

R 4 Requisitando um garfo f:

$Hungry, (f \in S), \sim FORK(f), req(f) \rightarrow$
$send(f, Request)$
$req(f) := FALSE$

R 5 Enviando o garfo f após comer:

$Thinking, (f \in S), FORK(f), req(f) \rightarrow$
$send(f, DirtyFORK)$
$FORK(f) := FALSE$
$strongreq(f) := FALSE$

R 6 Enviando o garfo f após comer, mas ainda com fome:

$Hungry, (f \in S), FORK(f), req(f), dirty(f)$
send$(f, DirtyFORK)$
send$(f, Request)$
$FORK(f) := FALSE$
$req(f) := FALSE$
$strongreq(f) := FALSE$

R 7 Enviando o garfo f antes de comer:

$Hungry, (f \in S), FORK(f), strongreq(f), \sim dirty(f) \rightarrow$
send$(f, CleanFORK)$
$FORK(f) := FALSE$
$req(f) := FALSE$
$strongreq(f) := FALSE$

R 8 Recebendo um garfo f:

receive$(CleanFORK(f)) \rightarrow$
$FORK(f) := TRUE$
$dirty(f) := FALSE$

R 9 Recebendo a requisição fraca de um garfo f:

receive$(Request(f)) \rightarrow$
$req(f) := TRUE$

R 10 Recebendo a requisição forte de um garfo f:

receive$(StrongRequest(f)), FORK(f) \rightarrow$
$req(f) := TRUE$
$strongreq(f) := TRUE$

R 11 Recebendo a requisição forte após já ter enviado o garfo:

receive$(StrongRequest(f)), \sim FORK(f) \rightarrow$
$SKIP$

Algoritmo do Árbitro

R 12 Recebendo o pedido de um garfo f:

receive($Request(f)$) $\rightarrow$
$req(f) := TRUE$
$strongreq(f) := TRUE$

R 13 Recebendo um garfo f sujo:

receive($DirtyFORK(f)$) $\rightarrow$
$OWNER := NIL$
$GiveLeastPriorityTo(f)$

R 14 Recebendo um garfo f limpo:

receive($CleanFORK(f)$) $\rightarrow$
$OWNER := NIL$
$req(f) := TRUE$

R 15 Enviando um garfo f:

$req(f), (f \in R), f = HighestPriorityRequest, OWNER = NIL \rightarrow$
send($f, FORK$)
$OWNER := f$

R 16 Pedindo a devolução de um garfo f sem prioridade:

$(f \in R), OWNER \neq NIL, req(f), req(OWNER),$
$Priority(OWNER) > Priority(f) \rightarrow$
send($OWNER, Request$)
$req(OWNER) := FALSE$

R 17 Pedindo a devolução de um garfo f com prioridade:

$(f \in R), OWNER \neq NIL, req(f), strongreq(OWNER),$
$Priority(OWNER) < Priority(f) \rightarrow$
send($OWNER, StrongRequest$)
$strongreq(OWNER) := FALSE$

Condições Iniciais: Qualquer disposição dos garfos e inicialização das tabelas de prioridades que faça H inicialmente acíclico. Se um garfo está com um cliente, então a requisição deve estar com o árbitro ou vice-versa.

4.3 A Lista de Prioridades

A situação inicial da lista de prioridades para cada nodo é facilmente definida, pois determina-se uma instância do sentido das arestas do grafo H na inicialização do algoritmo. A atualisação desta lista será explicada a seguir.

Quando um árbitro a envia um garfo a um cliente j, ($j \in R_a$), ele sabe que j eventualmente comerá, então sua prioridade passará a ser a menor de todas. a supõe, contudo, que j não conseguiu comer (age de maneira pessimista). Quando outro cliente k ($k \in R_a$) requisita o recurso de a, a irá requisitá-lo de j. Quando j enviá-lo, poderá ser como um garfo sujo ou limpo. Se for um garfo limpo, significa que j não conseguiu comer e, assim, sua prioridade não será diminuida. Se o garfo estiver sujo, então a prioridade de j será diminuida pois isto significa que ele já comeu.

Este procedimento tem o inconveniente de que, durante o tempo em que j já tenha comido mas a não tenha pedido o garfo de volta, a lista de prioridades estará errada (pois j estará com prioridade alta quando sua prioridade já é baixa). Isto pode gerar o recebimento de uma requisição fraca por j ao invés de uma requisição forte. Mas se j não estiver comendo, a requisição fraca será atendida imediatamente, pois o garfo estará sujo. Se j estiver comendo, logo após terminar a requisição será atendida. Desta forma a correção na lista de prioridades será feita automaticamente.

5 Prova do Algoritmo

Teorema 5 *Se existe um caminho ligando um vértice w a um vértice u em H, e $w \neq u$, então $Priority(w) > Priority(u)$.*

Prova Provemos por indução no tamanho do caminho que liga w a u. Seja a o árbitro entre u e w.

Se o caminho wu tiver comprimento 1 temos que, se o garfo wu estiver com u e estiver sujo, ou então estiver sujo e em trânsito de u para a, isto significa que u acabou de comer e, por isto, tem a menor prioridade entre todos os vértices arbitrados por a. Em particular, $Priority(w) > Priority(u)$. Se nenhuma das duas condições acima for satisfeita, então a definição do sentido das arestas do grafo H diz que $Priority(w) > Priority(u)$ (veja a seção 4.1).

Suponha então a proposição válida para caminhos com tamanho menor ou igual a n. Seja então um caminho ligando w a u com tamanho $n+1$, e o vértice v situado no caminho que liga w a u. O tamanho do caminho que liga w a v é menor que ou igual a n, assim como o tamanho do caminho que liga v a u. Através da hipótese de indução temos que:

$$Priority(w) > Priority(v), \quad Priority(v) > Priority(u)$$

Como a relação de > (maior que) é transitiva, temos que:

$$Priority(w) > Priority(u)$$

Lema 1 *O grafo H é acíclico*

Prova Inicialmente o grafo H é acíclico por construção.

A direção de uma aresta em H só pode ser afetada quando um filósofo suja seus garfos, ou por modificações em alguma das tabelas de prioridades.

O envio de um garfo sujo não altera o grafo H, posto que isto só é feito quando o garfo já estava sujo após o filósofo haver comido.

No instante em que um filósofo suja seus garfos, ele dirige todas as arestas a ele incidentes para si. Desta forma, como não há arestas saindo dele, nenhum ciclo poderá ter sido gerado.

Vamos mostrar agora que nenhuma modificação da tabela de prioridades de qualquer árbitro, pode alterar o grafo H. A única operação que pode ser feita nas tabelas de prioridade é o rebaixamento de um filósofo j para a posição de menor prioridade.

Seja k um nodo qualquer do sistema e a o árbitro onde foi feito o rebaixamento de j. Se k não for arbitrado por a, a relação de precedência entre j e k não é alterada pelo rebaixamento.

Suponhamos então que a seja o árbitro de j e k. Se antes do rebaixamento de j, tivessemos $Priority(k) > Priority(j)$, então o rebaixamento de j não alteraria a posição relativa dos dois nodos no grafo H e por conseguinte, o sentido da aresta jk em H permaneceria inalterado, não sendo criado nenhum ciclo.

Suponhamos agora, que tivessemos $Priority(k) < Priority(j)$ antes da operação de rebaixamento. Neste caso, o rebaixamento inverte a posição relativa dos dois nodos na tabela de prioridades. Isto, porém, não altera o grafo H, porque a operação de rebaixamento de j só ocorre quando a recebe um garfo sujo de j, significando havia um garfo sujo em trânsito de j para a, e que, pela cláusula 2 da definição do grafo H (veja a seção 4.1), k precedia j no grafo H imediatamente antes da chegada do garfo sujo a a, e que continua a preceder imediatamente após a recepção, agora pela cláusula 3. Nenhuma alteração foi feita em H.

Desta forma, como o grafo H inicialmente é acíclico e como nenhuma operação sobre H gera ciclos, temos que H é acíclico.

Lema 2 ***Um árbitro que receba infinitas vezes o seu recurso não pode preterir indefinidamente um cliente requisitante qualquer.***

Prova Seja c o cliente que se julga perseguido. A cada passagem do recurso pelo árbitro, ele poderá recebê-lo sujo ou limpo. Se estiver sujo, a prioridade do nodo que o entregou passará a ser inferior à de c e assim permanecerá até que c receba o garfo e coma. Se estiver limpo, é porque foi devolvido em reação a um *StrongRequest*, o que significa que haveria um outro cliente requisitante com prioridade maior. Para que c nunca ascenda à posição de maior prioridade é preciso portanto que tenhamos um número infinito de clientes distintos com prioridades crescentes, o que evidentemente não é possível. Logo, em tempo finito c terá a maior prioridade entre todos os nodos requisitantes e receberá o recurso ficando com ele até que use.

Teorema 6 *Todo filósofo faminto come após uma espera finita.*

Prova Seja v_0 um filósofo faminto. Se v_0 tem todos os garfos ele pode comer. Suponhamos então que falta a v_0 o garfo que ele compartilha com v_1. O árbitro a que administra este garfo está ou estará com a requisição de v_0. Vamos mostrar que a única possibilidade de v_1 não devolver o garfo em tempo finito ao árbitro implica em termos v_1 eternamente faminto e v_1 preceder v_0 no grafo H.

Suponhamos que $Priority_a(v_0) < Priority_a(v_1)$. Neste caso o árbitro a enviou ou vai enviar a v_1 uma requisição. Se v_1 não devolver em tempo finito o garfo ao árbitro, isto significa que seu garfo está limpo, e que portanto v_1 precede v_0 no grafo H.

Suponhamos que $Priority_a(v_0) > Priority_a(v_1)$. Neste caso o árbitro a enviou ou vai enviar a v_1 um *StrongRequest*, que forçará v_1 a devolver o garfo em tempo finito. De acordo com o lema 2, se v_1 entregar o garfo ao árbitro, então v_0 fatalmente irá recebê-lo.

Se v_1 permanece eternamente faminto, é porque lhe falta um de seus garfos que está com v_2, que precede v_1 no grafo H e que também permanece eternamente faminto. Desta forma, a única maneira de termos v_0 eternamente faminto é termos uma lista ordenada de vértices famintos de tal maneira que v_i precede v_{i-1} no grafo H.

Se algum filósofo desta lista estiver com todos os garfos (neste caso ele seria o último da lista), ele poderá comer e, após comer, de acordo com o lema 2, entregará o garfo ao seguinte da lista que então comerá. Desta forma todos vértices comeriam em tempo finito, em particular v_0.

Provemos então que ao menos um vértice da lista deve obrigatoriamente estar com todos os garfos.

Para que não houvesse um nodo que estivesse com todos os garfos só há duas possibilidades: Haver um número infinito de vértices, o que certamente não ocorre, ou haver um ciclo em H, o que não ocorre, segundo o lema 1. Desta forma, existe ao menos um vértice que está com todos os garfos que necessita.

Assim vemos que um vértice não pode ficar esperando indefinidamente por outro vértice. Assim, todo filósofo faminto come em tempo finito.

6 Complexidade

Considera-se, na discussão a seguir, que a troca de mensagens é feita de uma maneira otimizada, ou seja, sempre que se for enviar mais de um tipo de mensagem (por exemplo, um garfo e uma requisição) ao mesmo tempo, somente uma mensagem será efetivamente enviada.

Inicialmente, um filósofo i tem os garfos que divide com os vértices j tais que $j \in R_i$. Estes garfos controlam o recurso de i. Os garfos que divide com os vértices j tais que $j \in S_i$ não estão com i. Desta forma, ao requisitar a exclusão mútua pela primeira vez, i deve pedir todos estes garfos trocando assim $2\sqrt{n}$ mensagens, sendo $\sqrt{n}$ requisições e $\sqrt{n}$ respostas.

Após conseguir todos os garfos, i pode entrar na região crítica quantas vezes quiser desde que os garfos não sejam requisitados por outros nodos. Tem-se assim a resposta do algoritmo no melhor caso, que é de zero mensagens.

Esta localidade da posição dos garfos funciona não somente para um vértice como também para um grupo deles. Se apenas um conjunto de vértices estiver ativo, ou seja, requisitar a exclusão mútua, as mensagens, após algum tempo, serão trocadas apenas entre eles (e possivelmente alguns outros vértices que fazem o papel de árbitros). Se, por exemplo, apenas dois vértices vizinhos estiverem ativos, o número de mensagens trocadas entre eles a cada vez que obtiverem a exclusão mútua será de dois (uma requisição e um envio), fazendo uma média de uma mensagem por exclusão mútua. Se os vértices não forem vizinhos, este número sobe para quatro, pois aí estará envolvido também o vértice em que estará o árbitro entre os dois vértices. A média neste caso é de duas mensagens por exclusão mútua.

O pior caso em que um filósofo i desejá a exclusão mútua ocorre quando ele não está com nenhum dos recursos e os donos dos mesmos também não. Ou seja, $\forall j, (j \in S_i), FORK_i(j) = FALSE, OWNER_j \neq NIL$. Neste caso, i requisitará os garfos a todos os vértices $j \in S_i$ (enviando $\sqrt{n}$ *reqb*). Cada j, por sua vez, requisitará o garfo ao vértice $OWNER_j$. Quando todos os $OWNER_j$ devolverem os garfos, mais $2\sqrt{n}$ mensagens terão sido trocadas. Agora os garfos serão passados a i, gastando mais $\sqrt{n}$ mensagens. Assim, no pior caso, são enviadas $4\sqrt{n}$ mensagens.

Pode-se estimar um caso médio da seguinte forma: Dos $\sqrt{n}$ elementos de S_i, i tem os garfos de metade. Logo deve requisitar a outra metade enviando $\sqrt{n}/2$ mensagens. Destes $\sqrt{n}/2$ nodos, a metade terá emprestado seus recursos, ou seja, $\sqrt{n}/4$. Assim $2\sqrt{n}/4$ mensagens serão trocadas para se conseguir estes garfos. Mais $\sqrt{n}/2$ mensagens serão enviadas para remeter-se os garfos a i. No total, $1.5\sqrt{n}$ mensagens terão sido trocadas no caso médio.

7 Conclusão

O algoritmo aqui apresentado tem como principais vantagens o pequeno número de mensagens e o pequeno tamanho da mesma, permitindo, com sua implementação, uma economia no custo para se conseguir a exclusão mútua em sistemas distribuídos, aumentando assim sua

eficiência.

Aliando-se as principais vantagens dos dois conceitos aqui apresentados, conseguiu-se uma sensível redução nos números mínimo e máximo de troca de mensagens, prevendo-se, por isto que seu maior ganho de desempenho ocorrerá em situações de pequena ou alta demanda.

A simplicidade da mensagem (que pode ser codificada em apenas dois bits), permite inclusive que o algoritmo possa ser usado em protocolos de baixo nível, como por exemplo a disputa pelo barramento do computador entre os vários processadores. Neste caso o algoritmo pode ser útil para implementação de sistemas operacionais de supercomputadores com vários processadores.

Isto, porém, não quer dizer que o algoritmo tenha somente aplicações relacionadas diretamente a sistemas operacionais. Por sua grande versatilidade e generalidade, ele pode ser usado tanto em aplicações de hardware como em sistemas aplicativos, como por exemplo um sistema de reserva de passagens aéreas ou um sistema de controle bancário distribuídos.

8 Bibliografia

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

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

[3] Lakshman, T.V., and Agrawala A.K. *Efficient decentralized consensus protocols.* IEEE Trans. Soft. Eng., SE-12,5 (May 1986), 600–607.

[4] Ricart, G., and Agrawala, A.K. *An optimal algorithm for mutual exclusion in computer networks.* Commun. ACM 24,1 (Jan. 1981), 9–17.

[5] Thomas, R.H. *A majority consensus approach to concurrency control for multiple copy databases.* ACM Trans. Database Syst. 4,2 (June 1979), 180–209

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

[7] Hoare, C.A.R. *Communicating sequential processes.* Commun. ACM 21,8 (Aug. 1978), 666–677.

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

[9] Dijkstra, E.W. *Guarded commands, nondeterminancy and formal derivation of programs.* Commun. ACM 18,8 (Aug. 1975), 453–457.

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

[11] Albert, A.A., and Sandler, R. *An introduction to finite projective planes.* Holt, Rinehart, and Winston, New York, 1968.