Exclusão Mútua

Motivação

- Já conhecemos o problema da exclusão mútua em sistemas operativos
  - O objectivo é garantir que dois processos não acedem ao mesmo recurso (e.g., um ficheiro, uma estrutura de dados, um dispositivo de entrada de dados) ao mesmo tempo, o que poderia causar incoerências
- Em sistemas distribuídos o problema é o mesmo…
- Solução: **exclusão mútua distribuída**
  - Os processos necessitam de se coordenadar para garantir que não há dois processos a acederem simultanemanete ao recurso
  - Exclusão mútua baseada exclusivamente na **troca de mensagens**

Solução centralizada, cliente-servidor

- Um processo é eleito como servidor e controla uma ”chave” que dá acesso ao recurso.
- Um cliente que queira aceder ao recurso pede a chave ao servidor
- Se o servidor tiver a chave, dá a chave ao cliente, caso contrário coloca o cliente em lista de espera
- Quando um cliente devolve a chave, o servidor dá a chave ao próximo cliente na lista de espera (caso exista)

Solução centralizada, cliente-servidor

- O servidor pode ficar sobrecarregado
- Quando o servidor falha o sistema bloqueia
- A chave tem de passar sempre pelo servidor
  - O sistema seria mais eficiente se um cliente pudesse entregar a chave directamente ao próximo

- Conseguimos desenhar uma solução **descentralizada**?

Algoritmo de Ricart and Agrawala

- Algoritmo descentralizado
- Não é perfeito, longe disso:
  - Também não é tolerante a faltas
  - Em vez de sobrecarregar um processo especial (o servidor), sobrecarrega todos os processos
- Mas é interessante se for visto como um passo para atingir uma versão descentralizada (mas melhor)

- Permite um cliente passar a “chave” directamente a outro cliente!

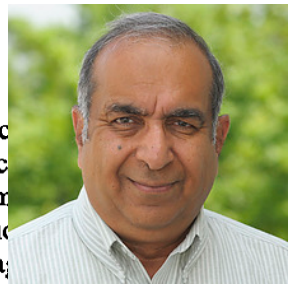
Algoritmo de Ricart and Agrawala

Operating Systems — R. Stockton Gaines, Editor

An Optimal Algorithm for Mutual Exclusion in Computer Networks

Glenn Ricart
National Institutes of Health

Ashok K. Agrawala
University of Maryland



1. Introduction

An algorithm is prop tual exc
sion in a computer netw nmunic
only by messages and do is assum
that there is an error-fi unicati
network in which transi l messa
may not be delivered in the order sent. Nodes are assumed to operate correctly; the consequences of node failure are discussed later. The algorithm is symmetrical, exhibits fully distributed control, and is insensitive to the relative speeds of nodes and communication links.

The algorithm uses only $2^*(N-1)$ messages between nodes, where N is the number of nodes and is optimal in the sense that a symmetrical, distributed algorithm cannot use fewer messages if requests are processed by each node concurrently. In addition, the time required to obtain the mutual exclusion is minimal if it is assumed that the nodes do not have access to timing-derived information and that they act symmetrically.

While many writers have considered implementation of mutual exclusion [2,3,4,5,6,7,8,9], the only earlier al-

Algoritmo de Ricart and Agrawala

- Algoritmo descentralizado
- Não é perfeito, longe disso:
  - Também não é tolerante a faltas
  - Em vez de sobrecarregar um processo especial (o servidor), sobrecarrega todos os processos
- Mas é interessante se for visto como um passo para atingir uma versão descentralizada (mas melhor)

- Permite um cliente passar a “chave” directamente a outro cliente!

Algoritmo de Ricart and Agrawala

On initialization
    state := RELEASED;

To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Request processing deferred here
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD *or* (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Retirado da página 653 do livro da cadeira!

Algoritmo de Ricart and Agrawala

- Podemos usar outro critério para desempate, em vez dos relógios lógicos de Lamport?
  - Por exemplo, a “prioridade” do processo?

Algoritmo de Maekawa

- Consegue distribuir a carga, isto é, não há um processo que receba todos os pedidos
- Processos organizados em subconjuntos processos de chamados **quóruns** (V_0, V_1, etc.)
  - Um processo pode pertencer a mais que um quórum
  - Para qualquer par de quóruns, a sua interceção não pode ser vazia ($V_i \cap V_j \neq \emptyset$)
- **Cada processo pode votar num pedido de entrada em secção crítica, mas nunca pode votar em mais que um pedido em simultâneo**

Propriedade fundamental:
Qualquer par de quóruns intercepta em pelo menos um processo, logo 2 pedidos concorrentes nunca conseguem, cada um, receber os votos de quóruns inteiros

- Sofre de interbloqueio

Algoritmo de Maekawa

A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems

MAMORU MAEKAWA
University of Tokyo

An algorithm is presented that uses only $c\sqrt{N}$ messages to create mutual exclusion in a computer network, where N is the number of nodes and c a constant between 3 and 5. The algorithm is symmetric and allows fully parallel operation.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*mutual exclusion*; C.2.1 [**Computer Systems Organization**]: Network Architecture and Design—*network communications* C.2.4 [**Computer Systems Organization**]: Distributed Systems—*network operat-*

Algoritmo de Maekawa

On initialization
    state := RELEASED;
    voted := FALSE;

For p_i *to enter the critical section*
    state := WANTED;
    Multicast *request* to all processes in V_i;
    Wait until (number of replies received = *K*);
    state := HELD;

On receipt of a request from p_i *at* p_j
    if (*state* = HELD *or voted* = TRUE)
    then
        queue *request* from p_i without replying;
    else
        send *reply* to p_i;
        voted := TRUE;
    end if

For p_i *to exit the critical section*
    state := RELEASED;
    Multicast *release* to all processes in V_i;

On receipt of a release from p_i *at* p_j
    if (queue of requests is non-empty)
    then
        remove head of queue – from p_k, say;
        send *reply* to p_k;
        voted := TRUE;
    else
        voted := FALSE;
    end if

Retirado da página 640 do livro da cadeira!

Qual a melhor solução?

Depende do critério…

- Qual a mais eficiente?
- Qual a mais escalável?
- Qual a mais tolerante a falhas?

Qual a mais eficiente?

Algoritmo	***Bandwidth usage*** **(total de mensagens trocadas entre *enter/exit* por um mesmo cliente)**	***Client delay*** **(tempo para um processo *entrar* em secção crítica livre)**	***Synchronization delay*** **(tempo entre *exit* por um processo e *enter* por outro que estava à espera)**
Centralizado	3	2	2
Ricart&Agrawala	2 x (N-1)	2	1
Maekawa	3 x quorum_size	2	2 (*) assumindo que os 2 quóruns intercetam em apenas 1 processo

Como distribuem a carga?

(Para conseguir escalabilidade)

- Centralizada: tudo passa pelo coordenador, que pode ficar sobrecarregado
- Ricart&Agrawala: todos são sobrecarregados!
- Maekawa: um pedido só afeta um subconjunto de processos (um quórum)

Que falhas toleram?

- Nenhuma tolera perdas de mensagem (assumem canal fiável)
- Centralizada: não tolera a falha do coordenador
  - mas tolera falha de cliente que não detenha nem tenha pedido o token
- Ricart&Agrawala: não toleram a falha de qualquer processo
- Maekawa: tolera a falha dos processos que não esteja no quórum

É possível adaptar cada solução para melhorar a sua tolerância a falhas, **usando técnicas que estudaremos mais à frente**