
# *Sistemas Distribuídos*

*Comunicação em grupo*

António Rui Borges

# Resumo

- *Caracterização do problema*
- *Acesso a um objeto compartilhado em exclusão mútua*
  - *Permissão de acesso centralizado*
  - *Anel lógico*
  - *Ordenação total de eventos*
  - *Minimizando o número de mensagens*
- *Procedimento eletivo*
  - *Eleição em um anel lógico*
  - *Eleição em um grupo não estruturado*
- *Leitura sugerida*

# *Caracterização do problema*

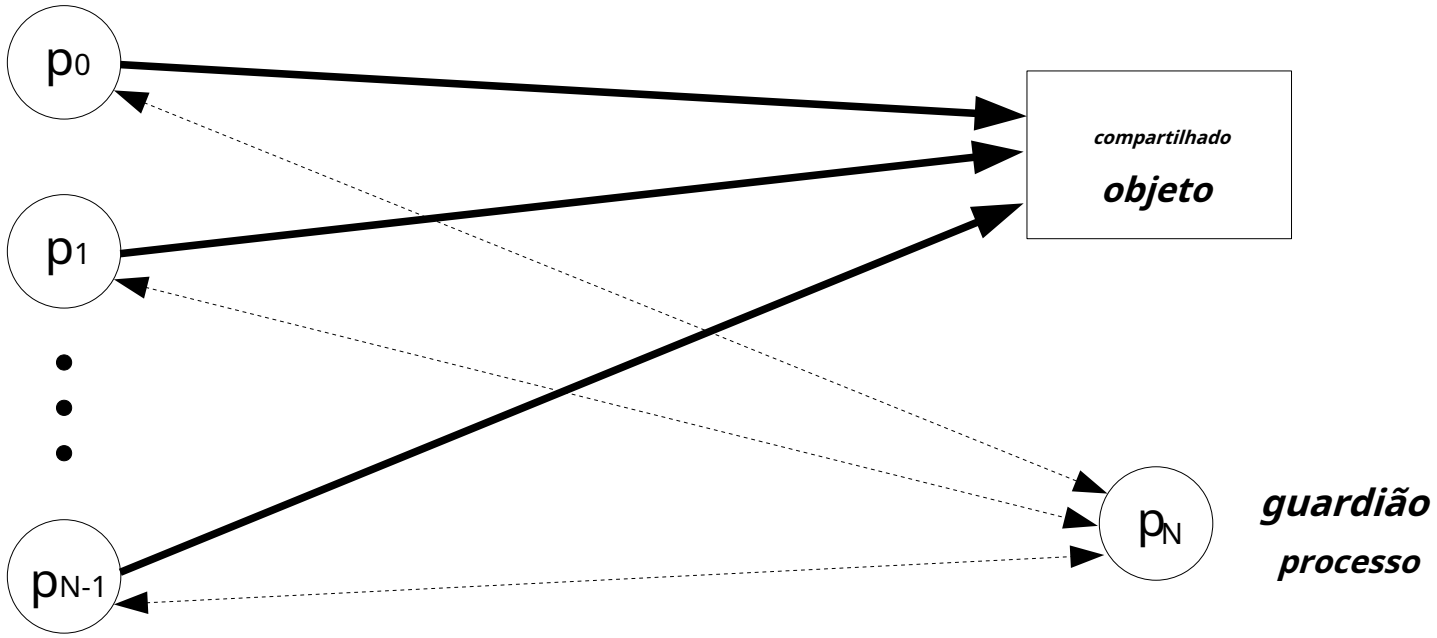
Em *comunicação em grupo*, todos os processos envolvidos têm a mesma posição. Pode-se presumir que não há nenhuma característica relevante que distinga uma das outras. Dizemos então que estamos numa situação de *comunicação entre iguais*.

Assim, quando acessam um recurso comum, devem ser concebidos meios para evitar condições de corrida que possam levar à inconsistência de informações. Como não compartilham em geral um espaço de endereçamento, a sincronização entre eles deve ser realizada por meio de passagem de mensagens.

Podemos supor por enquanto que

- as mensagens de troca têm tempo de transmissão finito, mas sem limite superior
- não há perda de mensagem.

# *Permissão de acesso centralizado - 1*




É uma adaptação quase direta do modelo cliente-servidor com*solicitar serialização*.

Existe um processo especial, o*processo de guardião*, que rastreia os acessos ao objeto compartilhado e permite acesso a ele por solicitação.

# *Permissão de acesso centralizado - 2*

**Protocolo**

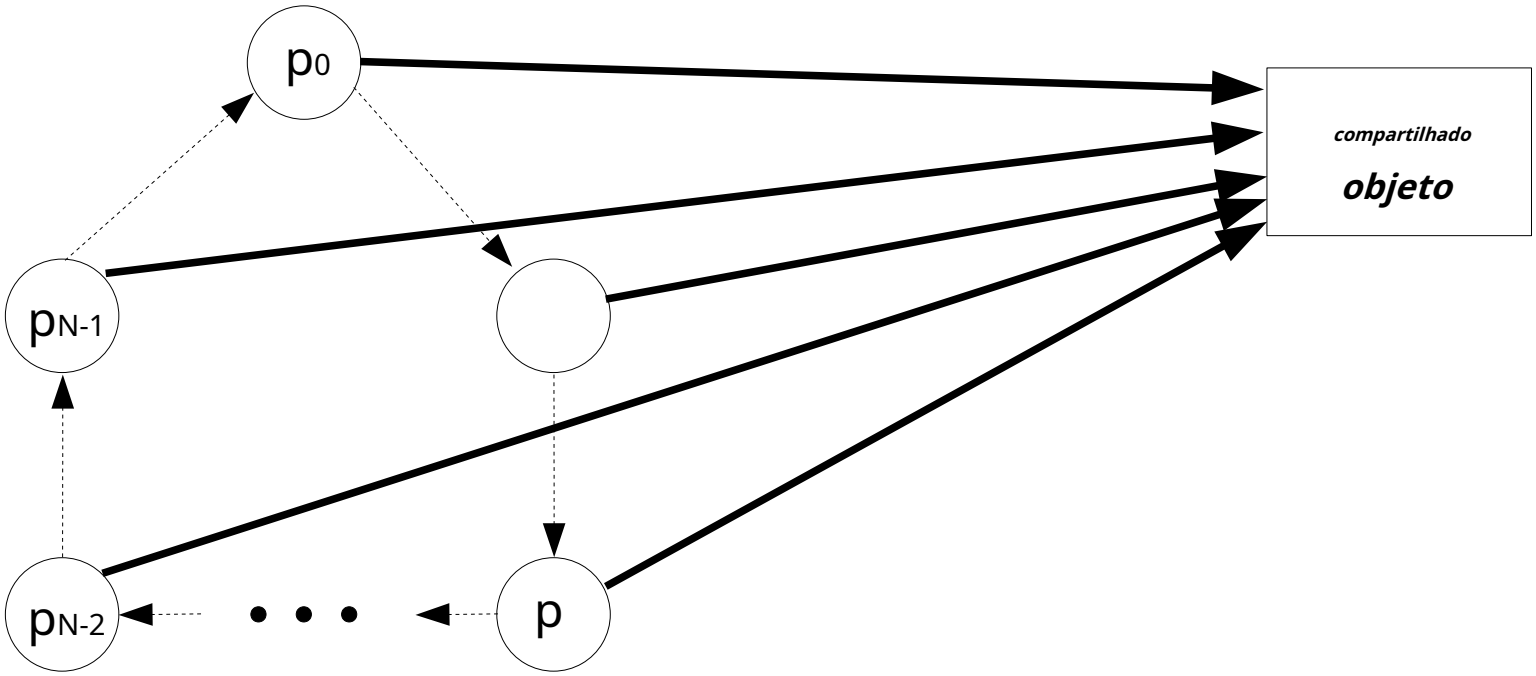
- sempre que um dos processos pares $p$, com $eu$=0, 1,-, N-1, deseja acesso ao objeto compartilhado, envia uma mensagem de *solicitar acesso* para $p_N$, solicitando permissão, e aguarda a recepção de uma mensagem de *garantir acesso* a partir dele
- se, no momento, não houver outro processo acessando o objeto compartilhado, o processo $p_N$ responde imediatamente; caso contrário, ele insere a solicitação em uma fila de espera
- quando a mensagem de *garantir acesso* é recebido, processa $p_{eu}$ pode acessar o objeto compartilhado
- quando o acesso ao objeto compartilhado for encerrado, processe $p_{eu}$ envia uma mensagem de *liberar acesso* para $p_N$, sinalizar que não requer mais o objeto
- se houver solicitações pendentes na fila de espera, processe $p_N$ recupera o primeiro e responde ao processo referenciado com *garantir acesso*.

# *Permissão de acesso centralizado - 3*

**Comentários**

- três mensagens são trocadas por acesso
- não é realmente uma verdadeira solução peer-to-peer, porque supõe a existência de um processo especial, o *processo de guardião*, para controlar o acesso ao objeto compartilhado
- assim, tem um *ponto único* de falha, se houver um mau funcionamento no processo $p_N$, toda a operação é interrompida.

# *Anel lógico - 1*




Uma restrição é imposta aos processos que formam o grupo: eles são organizados em circuito fechado no que diz respeito à comunicação. Processo $p_{eu}$, com $eu$=0, 1,-,N-1, só pode receber mensagens do processo $p_{(eu-1) \bmod N}$ e enviar mensagens para processar $p_{(eu+1) \bmod N}$.

A *mensagem simbólica* circula continuamente entre eles. O acesso ao objeto compartilhado só pode ser feito pelo processo que tomou posse da mensagem.

# *Anel lógico - 2*

**Protocolo**

- se o processo $p_{eu}$ necessita de acesso ao objeto compartilhado, aguarda a recepção da mensagem token e uma vez retida, acessa o objeto; após o término do acesso, ele envia a mensagem token para o próximo processo no anel
- se o processo $p_{eu}$ não necessita de acesso ao objeto compartilhado, ao receber a mensagem token, ele a envia imediatamente para o próximo processo do anel.

# *Anel lógico - 3*

**Comentários**

- sempre se troca uma mensagem, ou há acesso, ou não
- é muito eficiente se o grupo de processos for pequeno
- entretanto, se for muito grande, um determinado processo poderá ter que esperar muito tempo para poder acessar o objeto, mesmo que nenhum processo esteja fazendo isso no momento.

# *Ordenação total de eventos - 1*

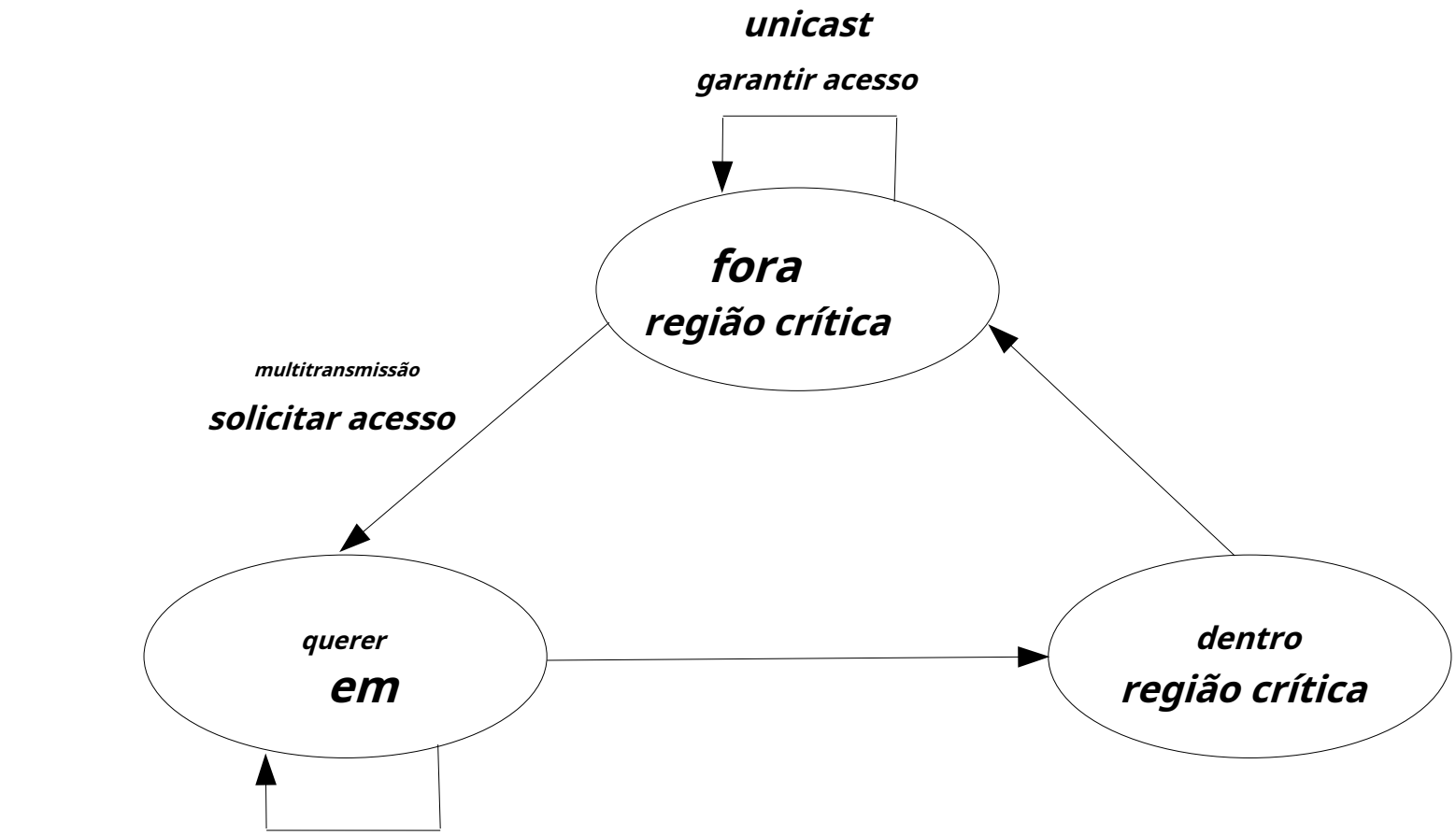
Ricart e Agrawala (1981) propuseram um método geral para garantir que N processos acessem um objeto compartilhado com exclusão mútua, ordenando totalmente suas solicitações de acesso através de um relógio lógico Lamport.

Assim, todos os processos ordenam da mesma forma os eventos associados aos pedidos de acesso ao objeto partilhado e atinge-se um consenso geral.

As mensagens trocadas incluem o carimbo de hora associado ao evento de envio, o que permite ao processo de recepção, no momento da recepção da mensagem, ajustar o seu relógio local de acordo com as regras prescritas por Lamport. Portanto, a causalidade potencial entre eventos pode ser explicitada.

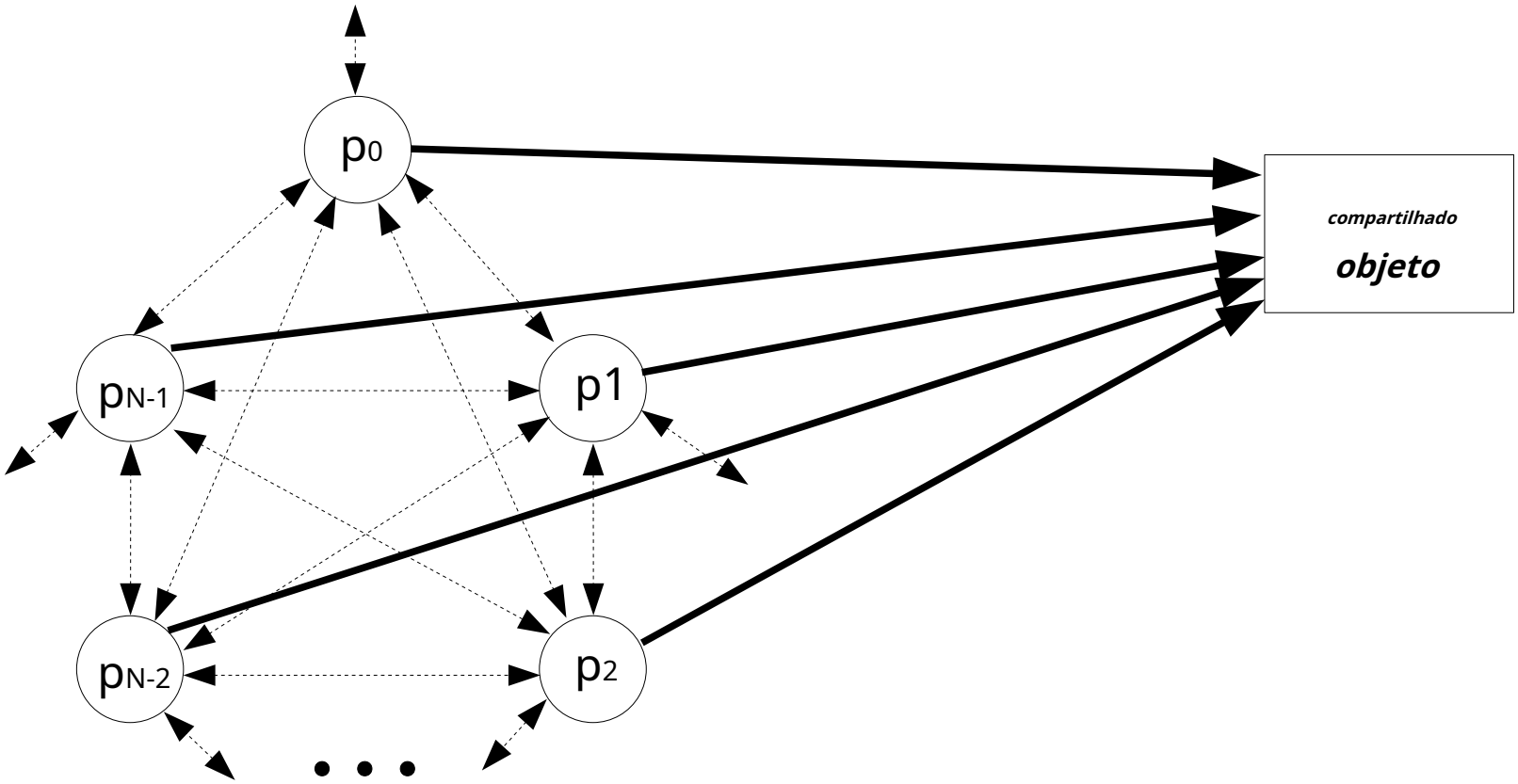
Para gerar a ordenação total dos eventos de solicitação de acesso, um carimbo de data/hora estendido contendo o par ordenado ($ts(eu_{ar}), eu\ ia(eu_{ar})$), onde $ts(eu_{ar})$ é o carimbo de data/hora da mensagem e $eu\ ia(eu_{ar})$ é a identificação do remetente, é construída.

# *Ordenação total de eventos - 2*




***aguarde a concessão de acesso de todos / acesso de concessão unicast***

# *Ordenação total de eventos - 3*



# *Ordenação total de eventos - 4*

***inicialização***
estado = foraCR;
requestMessageReady = **falso**;

***$p_{eu}$entra na região crítica***
estado = querIn;
númeroDeRequestsGranted = 0; minhaRequestMessage =
multicast (requestAccess); requestMessageReady =
**verdadeiro**;
**espere até**(numberOfRequestsGranted == N-1); estado =
dentro de CR;

***$p_{eu}$sai da região crítica***
estado = foraCR;
requestMessageReady = **falso**;
**enquanto**(!vazio (requestQueue))
{ id = getId (queueOut (requestQueue));
    unicast (id, acesso concedido);
}

# *Ordenação total de eventos - 5*

***$p_{eu}$recebe uma mensagem de solicitação de acesso de $p_j$***

```
se(estado == foraCR)
    { id = getId (requestMessage));
      unicast (id, acesso concedido);
    }
    senão se(estado == dentro de CR)
            queueIn (requestQueue, requestMessage); outro{
            espere até(requestMessageReady);
                se(getExtTimeStamp (minhaMensagemRequest) <
                        getExtTimeStamp (requestMessage)) queueIn
                (requestQueue, requestMessage); outro{id = getId
                (requestMessage));
                        unicast (id, acesso concedido);
                }
        }
```

***$p_{eu}$processa permissões de acesso***

```
númeroDeRequestsGranted += 1;
```

# *Ordenação total de eventos - 6*

**Comentários**

- 2(N-1) mensagens são trocadas por acesso
- é muito eficiente se o grupo de processos for pequeno
- entretanto, se for muito grande, muitas mensagens serão trocadas.

# *Minimizando o número de mensagens – 1*

Maekawa (1985) mostrou que o acesso em exclusão mútua a um objeto compartilhado por qualquer um dos processos pares existentes, não requer permissão de todos eles. Os processos pares podem ser organizados em grupos parciais e obter permissão apenas de todos os processos pertencentes ao seu grupo.

A exclusão mútua e, portanto, a eliminação das condições de corrida no acesso ao objeto partilhado são asseguradas desde que os gruposnão são mutuamente exclusivo.

O princípio subjacente é impor que um processo só acesse o objeto compartilhado se e somente se tiver permissão de todos os processos pertencentes ao seu grupo. A permissão é dada através de um processo de votação.

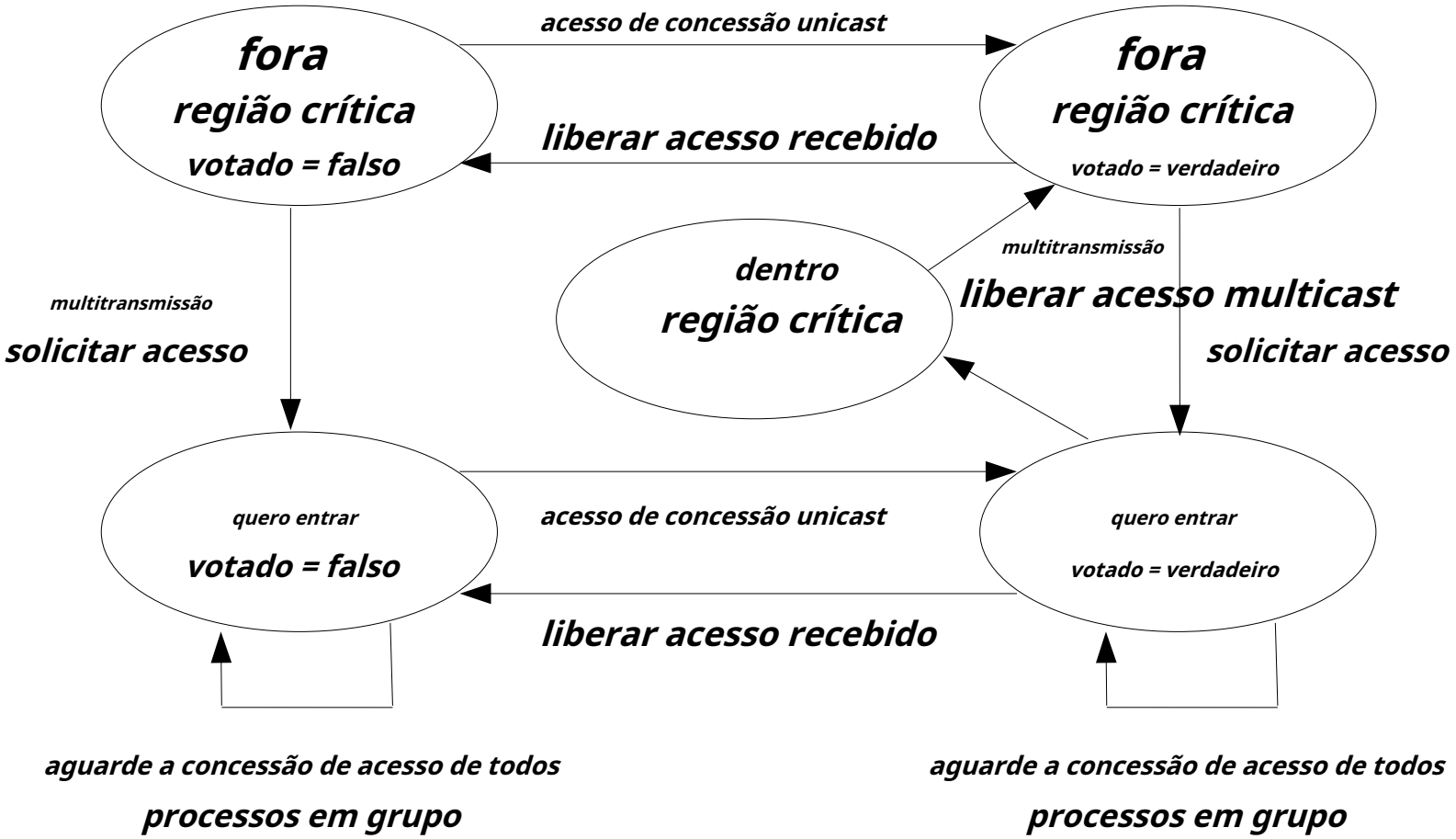
# *Minimizando o número de mensagens – 2*

Cada processo peer $p_{eu}$, com $eu$=0, 1,-, N-1, pertencem ao *grupo de votação* $V_{eu}$.

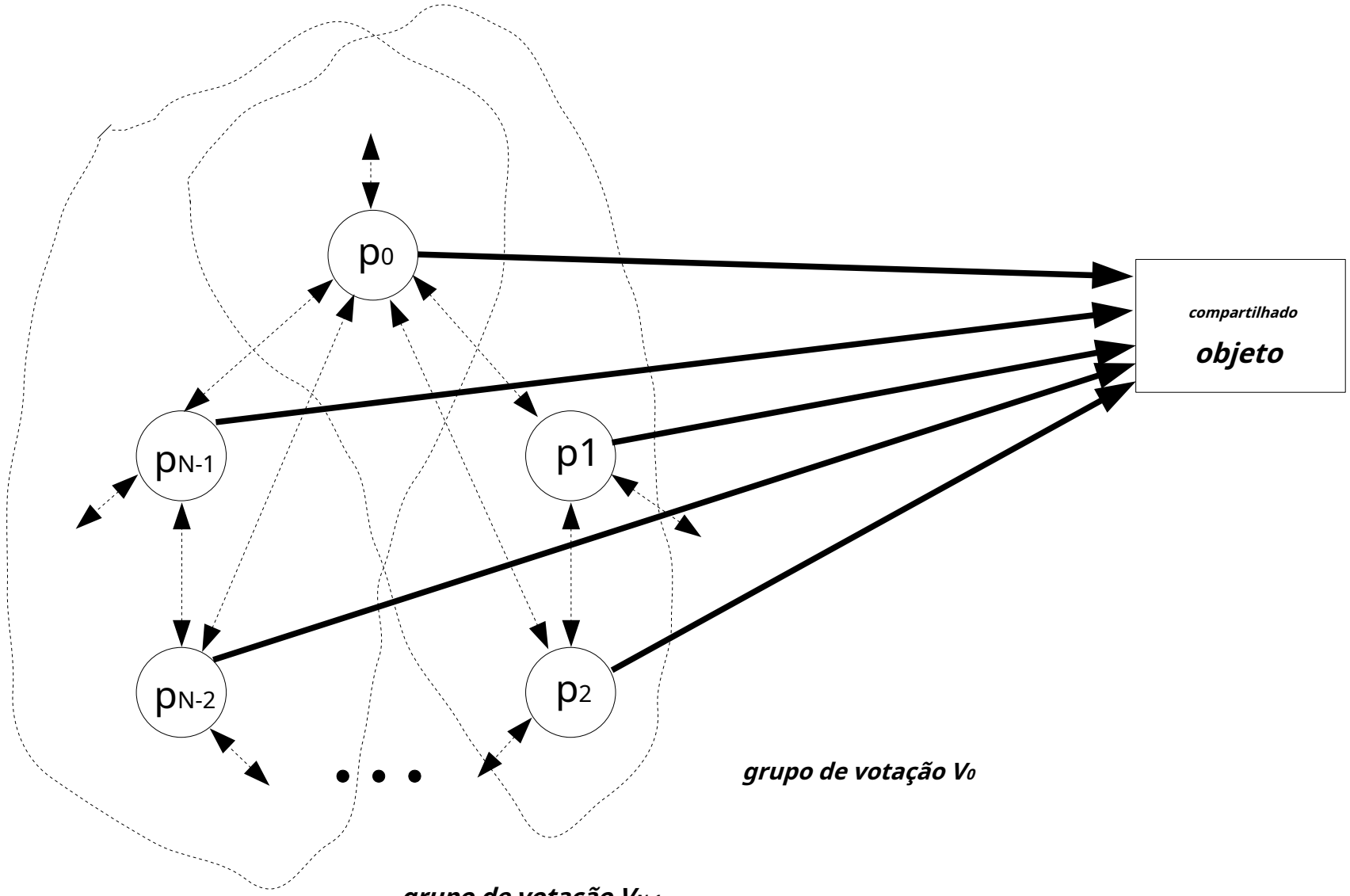
Os elementos de cada grupo de votação são escolhidos de forma que

- $\forall_{0 \leq eu < N}$ $V_{eu} \subseteq \{p_0, p_1, \cdots, p$ $\}$
- $\forall_{0 \leq eu < N}$ $p_{eu} \in V_{eu}$
- $\forall_{0 \leq eu\, j < N}$ $V_{eu} \cap V_j \neq \emptyset$
- $\forall_{0 \leq eu\, j < N}$ $\#(V_{eu}) \approx \#(V_j)$
- $\exists_{M \in \mathbb{N}} \forall_{0 \leq eu < N}$ $p_{eu}$ pertence a grupos M $V_*$.

# Minimizando o número de mensagens – 3



# *Minimizando o número de mensagens – 4*



# *Minimizando o número de mensagens – 5*

A determinação exata do conteúdo dos diferentes grupos de votação $V_{eu}$, com $eu$=0, 1,-, N-1 não é um procedimento simples. Existe, no entanto, uma aproximação que faz com que #($V_{eu}$) - #(Vj ),$M$- e é trivial.

| | | | | |
|---|---|---|---|---|
| | | … | $K$–2 | $K$–1 |
| $K$ | $K$+1 | … | 2$K$–2 | 2$K$–1 |
| … | … | … | … | … |
| ($R$–1)$K$ | ($R$–1)$K$+1 | … | $R.K.$–1 | 0 |

$V_1$= {0,1,⋯,$K$–1,$K$+1,⋯,($R$–1)$K$+1}

# *Minimizando o número de mensagens – 6*

***inicialização***
estado = foraCR;
votado =**falso**;

***$p_{eu}$entra na região crítica***
estado = querIn;
númeroDeRequestsGranted = 0;
multicast (requestAccess) para todos os processos em $V_{eu}$;
**espere até**(numberOfRequestsGranted == #($V_{eu}$)); estado =
dentro de CR;

***$p_{eu}$sai da região crítica***
estado = foraCR;
multicast (releaseAccess) para todos os processos em $V_{eu}$;

# *Minimizando o número de mensagens – 7*

***$p_{eu}$recebe uma mensagem de solicitação de acesso de $p_j$***

```
se((estado! = dentroCR) &&! votado)
      { id = getId (requestMessage));
          unicast (id, acesso concedido); votado
          =verdadeiro;
      }
      outroqueueIn (requestQueue, requestMessage);
```

***$p_{eu}$recebe uma mensagem de liberação de acesso de p***

```
se(!vazio (requestQueue))
      { id = getId (queueOut (requestQueue));
          unicast (id, acesso concedido); votado
          =verdadeiro;
      }
      outrovotado =falso;
```

***$p_{eu}$processa permissões de acesso***

```
númeroDeRequestsGranted += 1;
```

# *Minimizando o número de mensagens – 8*

**Comentários**

- 3ó(√N)mensagens são trocadas por acesso
- é muito eficiente quando o grupo de processos é muito grande.

# *Minimizando o número de mensagens – 9*

Este algoritmo está, no entanto, incorreto! Há casos em que, devido às condições prevalecentes da corrida, *impasse* pode ocorrer.

$V0 = \{0, 1\}$

$V1 = \{1, 2\}$

$V2 = \{2, 0\}$

Processos $p_0$, pág. 1 ep tente acessar o objeto compartilhado quase ao mesmo tempo. Pode acontecer que, devido aos momentos de chegada da mensagem, $p_0$ vota para seu próprio acesso, $p_1$ vota para seu próprio acesso e $p_2$ vota para seu próprio acesso. Assim, todos os processos recebem uma primeira permissão, mas nunca um segundo.

***Como resolver o problema?***

## *Minimizando o número de mensagens – 10*

Saunders (1987) mostrou que uma solução possível é ordenar totalmente as solicitações de acesso para evitar contenção. Outra é negar a condição de *espera circular*, ordenando as solicitações de acesso pela identificação do processo.

***Adapte o algoritmo Maekawa para evitar impasses!***

# *Procedimento eletivo - 1*

Há casos que exigem a seleção de um processo de um grupo para realizar uma tarefa bem definida em algum momento específico. Como todos os processos devem ser conceitualmente semelhantes e não podem ser distinguidos uns dos outros, qualquer um deles pode, em princípio, ser selecionado.

Pontos importantes para o estresse são

- o procedimento eletivo deve ser realizado em um número finito de etapas
- não pode haver ambigüidade, ou seja, apenas um processo é selecionado no final
- a decisão deverá ser consensual, ou seja, todos os processos envolvidos aceitarão a seleção.

# *Procedimento eletivo - 2*

Será assumido que

- o número de processos no grupo é fixo e previamente conhecido
- seu estado em um determinado momento é *em execução* ou *falha catastrófica*
- o tempo de transmissão das mensagens trocadas é finito, mas possui um limite superior, ou seja, *intervalos* pode ser definido
- mensagens podem ser perdidas.

# *Eleição em um anel lógico - 1*

- para começar, nenhuma eleição está ocorrendo e todos os processos estão em estado de *nenhum participante*
- qualquer processo poderá dar início ao procedimento eletivo; ele muda seu estado para *participante* e envia para o próximo processo no anel uma mensagem do tipo *começar a eleição* com identificação própria
- quando um processo recebe uma mensagem do tipo *começar a eleição*, ele realiza as seguintes ações
  - se a identificação do processo na mensagem for menor que a sua, ele envia a mensagem para o próximo processo no anel e muda seu estado para *participante*, se ainda não foi alterado
  - se a identificação do processo na mensagem for maior que a sua e ainda não for participante, ele substitui a identificação do processo pela sua própria, envia a mensagem para o próximo processo no anel e muda seu estado para *participante*; no entanto, se já estiver no estado de *participante*, descarta a mensagem para reduzir o número de mensagens circulantes
  - se a identificação do processo na mensagem for igual à sua, então o procedimento eletivo foi encerrado e o próprio processo foi eleito como *líder* (**por que?**)

# *Eleição em um anel lógico - 2*

- quando um processo é eleito como *líder*, ele envia uma mensagem do tipo *eleito* com identificação própria
- qualquer processo que receba uma mensagem do tipo *eleito* realiza as seguintes ações
  - ele muda seu estado para *nenhum participante*
  - se a identificação do processo na mensagem for diferente da sua, salva a identificação do *líder* processo e envia a mensagem para o próximo processo no anel
  - se a identificação do processo na mensagem for igual à sua, descarta a mensagem que leva ao final do procedimento eletivo.

# *Eleição em um anel lógico - 3*

Este algoritmo foi proposto por Chang e Roberts (1979) e resolve o problema se não houver falhas.

O que deve ser feito se o anel lógico necessitar de uma reconfiguração dinâmica devido a

- um processo mudando seu estado de *em execução* para *falha catastrófica*, ou vice-versa?
- perda de mensagem?

# *Eleição em grupo não estruturado – 1*

- para começar, nenhuma eleição está ocorrendo e todos os processos estão em estado de *nenhum participante*
- qualquer processo poderá dar início ao procedimento eletivo; ele muda seu estado para *participante* e envia para todos os processos com número de identificação menor uma mensagem do tipo *começar a eleição* com identificação própria
- quando um processo recebe uma mensagem do tipo *começar a eleição*, ele realiza as seguintes ações
  - ele responde ao processo remetente com uma mensagem do tipo *reconhecer*
  - se o seu estado fosse *nenhum participante*, ele muda seu estado para *participante* e envia para todos os processos com número de identificação menor uma mensagem do tipo *começar a eleição* com identificação própria
- quando um processo recebe uma mensagem do tipo *reconhecer*, ele aguarda uma mensagem do tipo *eleito* com a identificação do líder
- se, durante um determinado período de tempo, um processo tendo o *participante* estado não recebe nenhuma mensagem do tipo *reconhecer*, considera-se o processo com a identificação inferior que está vivo e assume o papel de *líder*, ele então envia uma mensagem do tipo *eleito* com identificação própria a todos os processos do grupo para informá-los do fato.

# *Eleição em grupo não estruturado – 2*

Este algoritmo foi proposto por Garcia-Molina (1982) e resolve o problema se não houver falhas.

O que deve ser feito se o grupo não estruturado necessitar de uma reconfiguração dinâmica devido a

- um processo mudando seu estado de *em execução* para *falha catastrófica*, ou vice-versa?
- perda de mensagem?

# *Leitura sugerida*

- *Sistemas Distribuídos: C uma vez n, 4ªEdi ção, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Capítulo 12: *Coordenação e acordo*
    - Seções 12.1 a 12.3
- *Sistemas Distribuídos: Princípios e Paradigmas, 2ªEdição,* Tanenbaum, van Steen, Pearson Education Inc.
  - Capítulo 6: *Sincronização*
    - Seções 6.3 a 6.5