Fundamentos de sistemas distribuídos

Coordenação

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…
  - Como garantir que processos em máquinas diferentes possam aceder a um mesmo **recurso partilhado**?
    - Como garantir **coerência** do recurso partilhado?
    - Problema da **secção crítica**
- Solução: **exclusão mútua distribuída**
  - Exclusão mútua baseada exclusivamente na **troca de mensagens**

Definição do problema e propriedades

Definição do problema

- Um conjunto de processos tenta aceder a um **recurso partilhado**
- O recurso só pode ser acedido por **um processo de cada vez**
- O acesso ao recurso é representado por uma **secção crítica**

Propriedades a serem satisfeitas (simplificadas)

- **Segurança (*Safety*)** – *"something 'bad' will never happen"*

  S1: Nunca há mais do que um processo na secção critica

- **Vivacidade (*Liveness*)** – *"something 'good' will happen (but we don't know when)"*

  V1: Se há um conjunto de processos a tentar aceder à secção critica, então um destes processos vai conseguir aceder

  V2: Qualquer processo que tenta aceder a secção crítica vai conseguir aceder (mais tarde ou mais cedo)

Safety* e *Liveness
são conceitos fundamentais em qualquer algoritmo de coordenação

Como resolver o problema?

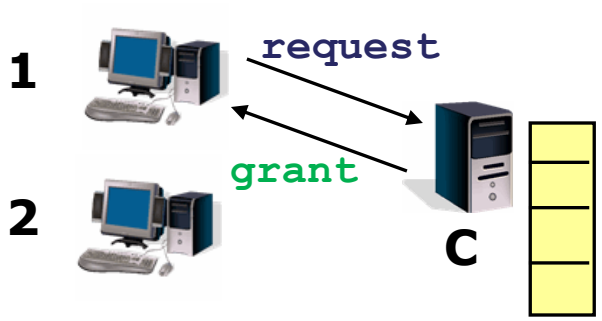
- Tipos de algoritmos de exclusão mútua em sistemas distribuídos
  - **Baseados em permissão**: o processo que tem permissão dos outros (ou do sistema) acede ao recurso
  - **Baseados em testemunho (*token*)**: o processo que tem o testemunho acede ao recurso

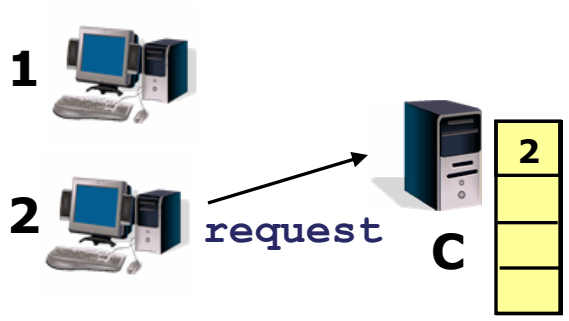
- Veremos 4 algoritmos que tentam resolver o problema
  1. Centralizado
  2. Descentralizado
  3. Distribuído
  4. Baseado em anel

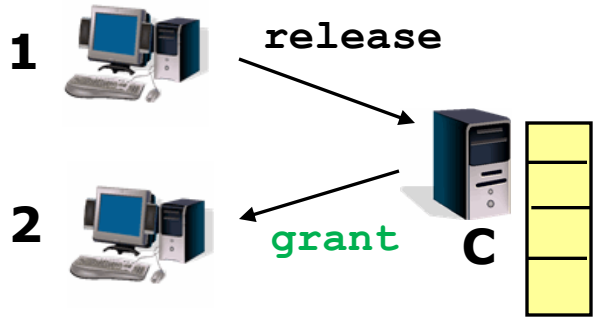
Algoritmo centralizado

- Pressupostos:
  - Os processos não falham e as mensagens não são perdidas
  - **Um** dos processos é seleccionado como **coordenador** (usando, p. ex., um dos algoritmos de eleição de líder que veremos em breve)
- Algoritmo:
  1. Sempre que um processo quer entrar numa região crítica, envia uma mensagem ao coordenador (`request`) com o identificador do recurso pedindo permissão para aceder
  2. O coordenador devolve uma mensagem (`grant`) indicando que o processo pode continuar (`OK`), se nenhum outro processo estiver nesse momento na região crítica
  3. Caso contrário, o coordenador não devolve qualquer mensagem (ou retorna uma mensagem `NOK` indicando que o processo não tem permissão), e coloca o processo numa **fila de espera**
  4. Quando um processo deixa a região crítica, envia uma mensagem ao coordenador libertando o recurso (`release`)
     a. Se houver pedidos na fila de espera, coordenador dá permissão ao pedido no topo da fila.

Algoritmo centralizado: prós e contras







- Propriedades satisfeitas:
  - **S1:** coordenador só deixa um processo usar o recurso de cada vez
  - **V1:** se há pedidos, um está sempre a ser satisfeito
  - **V2:** equitativo pois os pedidos são satisfeitos na ordem em que chegam e nenhum processo espera para sempre.
- **Vantagens:** o algoritmo é **equitativo** (satisfaz V2); **fácil** de implementar; necessárias apenas **duas mensagens** para aceder ao recurso (`request, grant`) e apenas três para todo o processo (`request, grant, release`)
- **Problemas:** o coordenador é um **ponto único de falha**; o coordenador pode trazer problemas de desempenho (***bootleneck*** do sistema)

Comparação entre algoritmos para exclusão mútua

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2

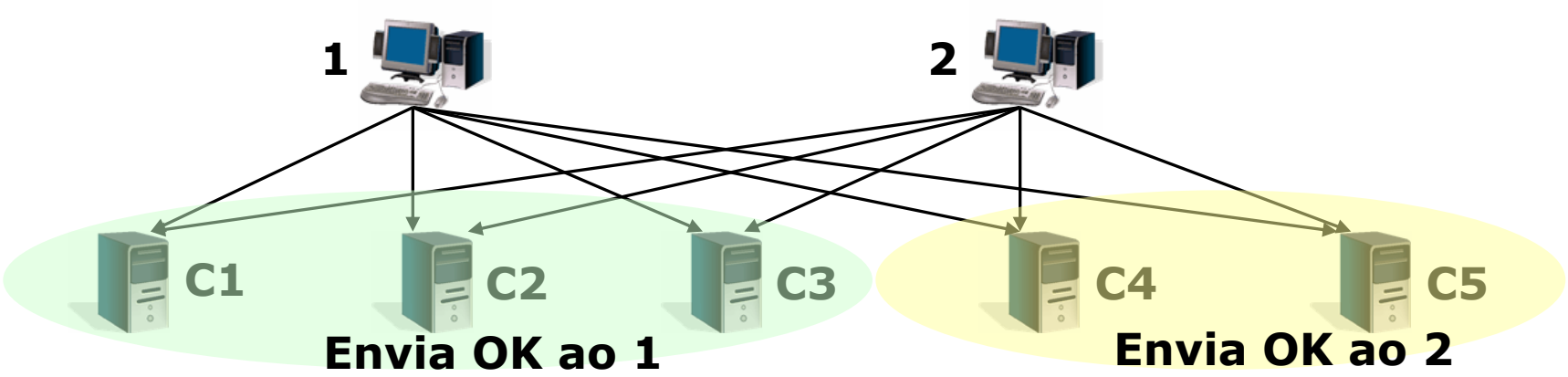
Algoritmo descentralizado

Lin [2004]
(pág. 327 van Steen)

- Pressupostos:
  - Os processos não falham e as mensagens não são perdidas
  - Há ***m* processos coordenadores**

- Algoritmo:
  1. Sempre que um processo quer entrar numa região crítica, envia uma **mensagem aos coordenadores** com o identificador do recurso a ser acedido
  2. Cada **coordenador** devolve uma mensagem **OK** indicando que o processo pode continuar, ou uma mensagem **NOK** indicando que o recurso está cedido a outro processo
  3. Um processo entra na **secção crítica** se recebe ***n > m/2* mensagens OK** de diferentes coordenadores
  4. Se a **maioria** das mensagens for **NOK**, o processo **avisa aos coordenadores que votaram OK** que ele não tem acesso **e espera** uma quantidade de tempo aleatória para voltar a tentar executar o algoritmo
     - Como o *backoff* usado para transmissão de dados nas redes Ethernet

Algoritmo descentralizado: prós e contras



- Propriedades satisfeitas:
  - **S1:** no máximo só um processo é que recebe uma maioria de "votos OK"
- **Vantagens:** é **fácil** de implementar; funciona em sistemas de **larga escala** (por exemplo sistemas P2P); **pode tolerar faltas** se aumentarmos o numero de votos OK necessários para se aceder a um recurso
- **Problemas:** Não satisfaz as propriedades de ***liveness*** V1 e V2
  - Quando há muitos processos a tentarem aceder a um recurso torna-se difícil ter maioria de "votos OK" e a utilização do recurso baixa

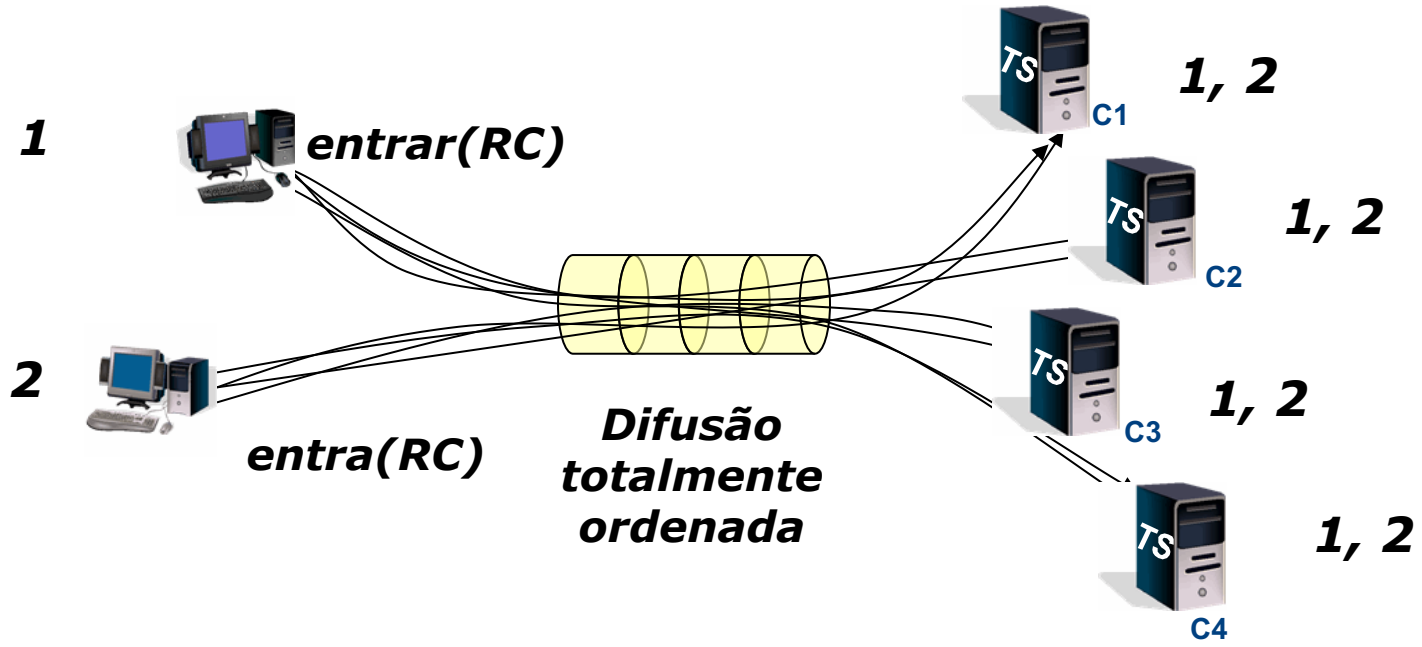
Comparação entre algoritmos para exclusão mútua

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Decentralized		

Algoritmo descentralizado: como satisfazer V2?

~Maekawa [1985]
(pág. 640 Coulouris)

- Se usarmos **difusão totalmente ordenada** para enviar os pedidos de acesso à região crítica (RC) e libertar os recursos dos processos aos coordenadores
  - Difusão totalmente ordenada: todos recebem a mesma sequência de mensagens
- No resto, o algoritmo é igual... e **satisfaz V1 e V2**



Algoritmo distribuído

Ricart and Agrawala [1981]
(pág. 637 Coulouris)

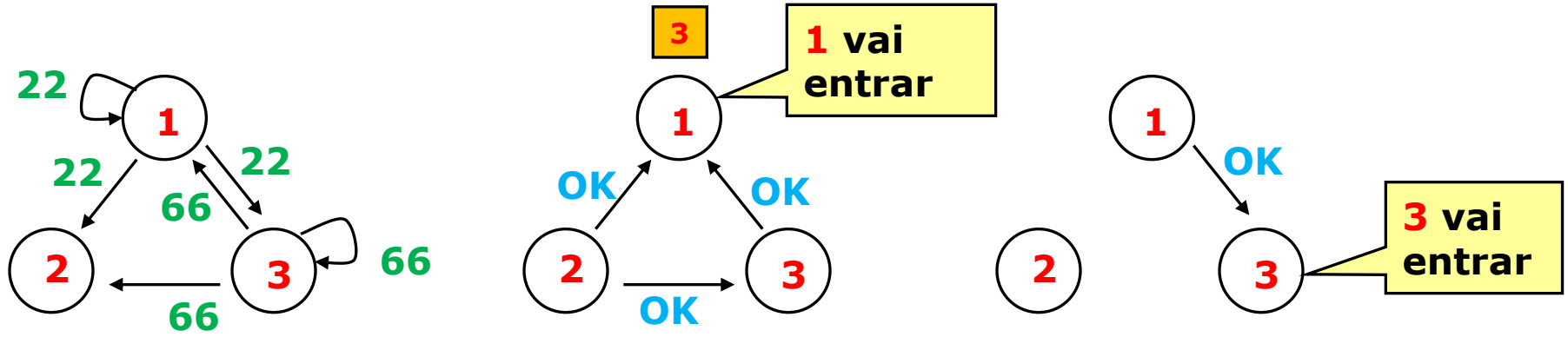
- Pressupostos:
  - Os processos não falham e as mensagens não são perdidas
  - Assume-se que **é possível ordenar todos os eventos no sistema** *(com **relógios lógicos**)*

- Algoritmo:
  - Quando um processo quer entrar numa região crítica **envia uma mensagem a todos os outros**, com a seguinte informação

    `<ID do processo;carimbo temporal>`

  - Quando um processo recebe a mensagem faz o seguinte
    1. devolve **OK** se não está na região crítica e não pretende entrar, *ou*
    2. se estiver ele próprio na região crítica **não responde e guarda** a mensagem numa fila de espera, *ou*
    3. se **também quiser entrar** na região crítica e ainda não tiver feito o pedido compara os carimbos temporais do seu pedido e da mensagem e
       - devolve **OK** se o da mensagem for menor ("**desiste**"), *ou*
       - **guarda a mensagem e não responde** se a do seu pedido for menor
  - O processo **só entra** na região crítica quando receber um **OK de todos** os processos
  - Quando o processo **deixa** a região crítica envia um **OK a todos** os processos que tenham mensagens na **fila** e depois apaga todas essas mensagens da fila

Algoritmo distribuído: prós e contras



- Propriedades:
  - **S1:** processo só acede quando recebe OK de todos os outros, por isso só o processo que tem o menor valor do relógio lógico é que entra (os outros que também querem cedem a sua vez). Em caso de "empate", usa-se ID do processo para desempatar.
  - **V1:** se há pedidos, um está sempre a ser satisfeito
  - **V2:** equitativo pois os pedidos ficam em fila de espera e são satisfeitos na ordem temporal lógica, por isso nenhum processo espera para sempre.
- **Vantagens:** o algoritmo é **equitativo** (satisfaz V2); não precisa de **coordenadores**
- **Problemas: mais mensagens** para entrar na região crítica; temos ***n* pontos de falha**; cada processo precisa de **manter lista** de todos os processos (*multicast* resolve este problema *[Ricart and Agrawala, 1981]*); *n* pontos em que podem surgir problemas de **desempenho**.

Comparação entre algoritmos para exclusão mútua

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed		
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

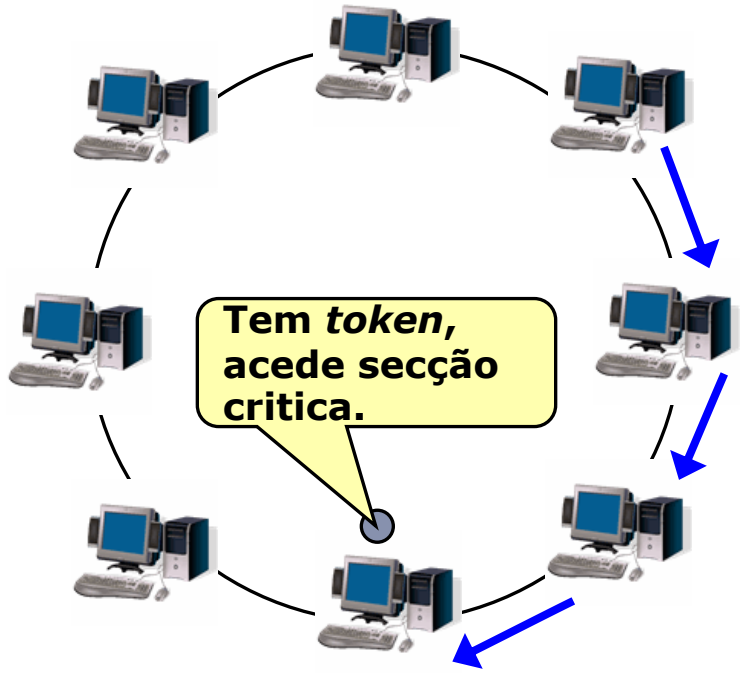
Algoritmo baseado em anel

- Pressupostos:
  - Os processos não falham e as mensagens não são perdidas
  - Utilizam-se os **identificadores** dos processos para os organizar num **anel lógico**

- Algoritmo:
  - Quando o anel é iniciado, associa-se ao processo 0 **um testemunho (*token*)**
  - O **testemunho vai circulando**, sendo trocado entre os processos pela ordem lógica definida pelo anel
  - Um processo apenas pode entrar numa **região crítica** quando **possuir o testemunho**
  - Um processo deve **passar o testemunho quando sai** da região crítica
  - O testemunho **continua a ser trocado** entre os processos ainda que nenhum deles queira entrar numa região critica

Algoritmo baseado em anel: exemplo

- Propriedades:
  - **S1:** Só o processo que tem o *token* é que pode aceder.
  - **V1:** O *token* circula, portanto se há pedidos, um está sempre a ser satisfeito
  - **V2:** É equitativo porque o *token* vai circulando pelo anel.



- **Vantagens:** o algoritmo é **equitativo** (satisfaz V2); entre **1** e *n* mensagens para se entrar na região crítica
- **Problemas: perca** do testemunho (*token*); **falha** de um processo; **requer envio** de mensagens (*token*) ainda que nenhum processo queira entrar numa região crítica

Comparação entre algoritmos para exclusão mútua

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$2 \cdot (N-1)$	$2 \cdot (N-1)$
Token ring		
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

Eleição de líder

Algoritmos de eleição de líder

- Muitos algoritmos distribuídos necessitam de seleccionar um processo de entre um conjunto de processos iguais: um coordenador ou um **líder**.

- Esta tarefa parece simples à partida, mas na prática é necessário ter algumas precauções porque alguns dos processos podem falhar (durante ou antes da execução do algoritmo)
  - E é fundamental que todos os processos **concordem** na escolha do líder

- Assume-se normalmente que:
  - cada processo tem um identificador único
  - **os processos podem falhar**
    - Em muitos sistemas reais, tem de se eleger um líder justamente porque o antigo não está a responder

Problema da eleição de líder

- Definição:
  - Existe um sistema com *n* processos;
  - Cada processo tem um identificador único;
  - Para eleger um líder, cada processo invoca a função `lider()`, que retorna o identificador do processo eleito como líder.

- Propriedades:
  - Segurança (*Safety*):

    S: Após a execução do algoritmo, existe **apenas um líder** que é conhecido por todos.
  - Vivacidade (*Liveness*):

    V: A execução do **algoritmo termina**.

- Algoritmos clássicos para eleição de líder
  - Algoritmo “bully” (algoritmo do “mandão”) [Garcia-Molina1982]
  - Algoritmo baseado em anel

Algoritmo “bully”

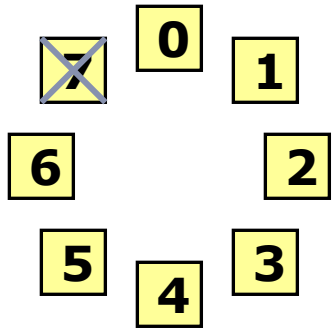
Pressupostos: há tempos máximos conhecidos para a comunicação; canais fiáveis

Algoritmo:

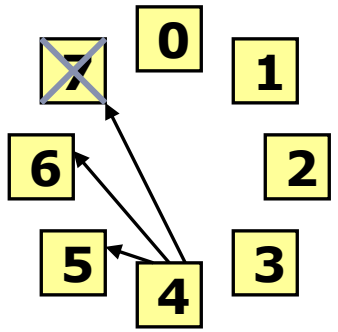
1. Um processo **P** inicia o algoritmo quando **deteta que o líder não responde** (esta deteção pode ser feita através de um *timeout*, por exemplo)
2. O processo **P** efetua os seguintes passos numa eleição:
   a) envia a mensagem **ELEIÇÃO** aos processos **com identificadores superiores**
   b) se **ninguém responde**, P ganha a eleição e fica como **líder**
   c) se um processo com identificador superior responde **OK**, **P pára** de executar o algoritmo
3. Se um processo recebe **ELEIÇÃO** de um processo com identificador inferior, responde com uma mensagem **OK** e executa o algoritmo de eleição (a não ser que já tenha iniciado)
4. Quando um processo **percebe que vai ser o próximo líder** (i.e., não recebe **OK** de nenhum processo com identificador superior a ele), envia uma mensagem **COORDENADOR** a **todos** os processos
5. Quando um **processo que estava parado entra** em funcionamento, **executa** o algoritmo
   – Se tiver o número mais alto, passa a ser coordenador. Daí ser o **algoritmo do “mandão”**

1. Algoritmo “bully”: exemplo

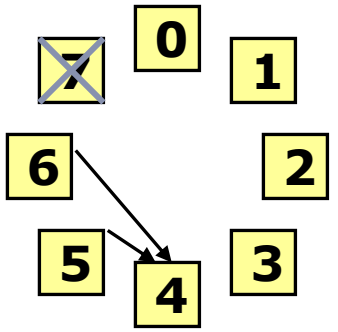
Líder falha



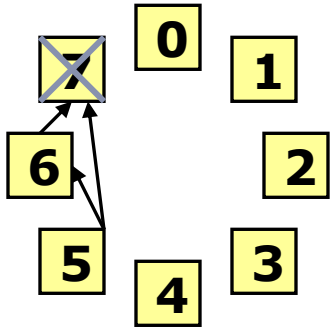
4 deteta falha do 7 e envia `ELEIÇÃO`



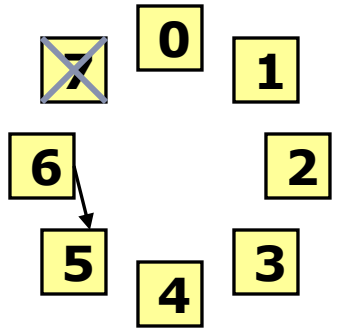
5 e 6 enviam `OK`



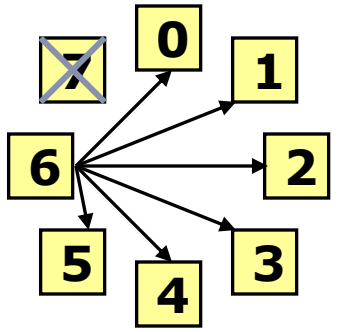
5 e 6 enviam `ELEIÇÃO`



6 envia `OK`



6 envia `COORDENADOR`



Prova de correcção do algoritmo “bully”

L: A execução do algoritmo termina.

- Quando um processo P começa a correr o algoritmo ele só pode ficar **bloqueado se algum** dos processos com identificador superior a P **não responderem** à sua mensagem de **`ELEIÇÃO`**.
- Como **há tempos máximos** conhecidos para a comunicação, um processo que não responde considera-se que falhou. Logo o **processo não bloqueia** e o algoritmo termina.

S: Após a execução do algoritmo existe apenas um líder e é conhecido por todos

Primeira parte – existe apenas um líder:

- Por ***redução ao absurdo***, assuma que o algoritmo falha a propriedade *S* e **dois processos P e Q são líderes**
- Isto significa que **quer P quer Q não receberam mensagens `OK`** de processos com identificadores maiores que o seu;
- Como os **identificadores são únicos e diferentes**, temos **P>Q ou Q>P**:
  - P>Q: então **Q recebeu a mensagem OK de P** em resposta à sua mensagem **`ELEIÇÃO`** e portanto não se declarou líder;
  - Q>P: então **P recebeu a mensagem OK de Q** em resposta à sua mensagem **`ELEIÇÃO`** e portanto não se declarou líder.
- Em ambos os casos **caímos numa contradição**, logo não pode haver dois processos líderes.

Segunda parte – todos sabem quem é o líder:

- Antes de o algoritmo terminar o líder envia a **mensagem `COORDENADOR` para todos** os outros processos.
- Como os **canais são fiáveis**, todos recebem essa mensagem e ficam a conhecer a identidade do novo líder.

Algoritmo baseado em anel

Pressupostos: há tempos máximos conhecidos para a comunicação; canais fiáveis

Algoritmo: os processos encontram-se organizados num círculo lógico (um **anel**)

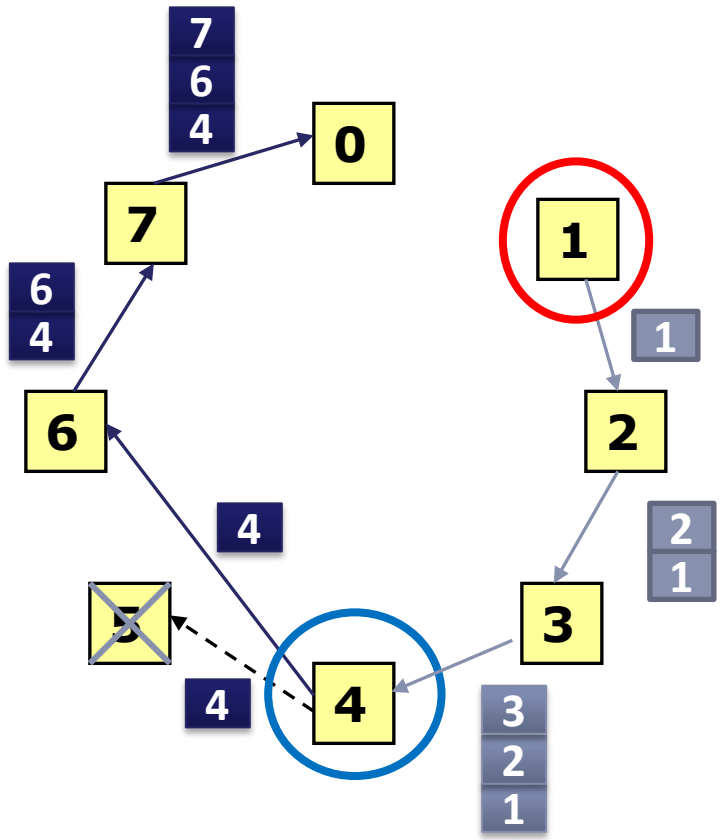
- Quando um processo P detecta que o **líder não se encontra em funcionamento**, constrói uma mensagem **`ELEIÇÃO`** e envia-a ao seu sucessor

  `ELEIÇÃO = < identificador do processo >`

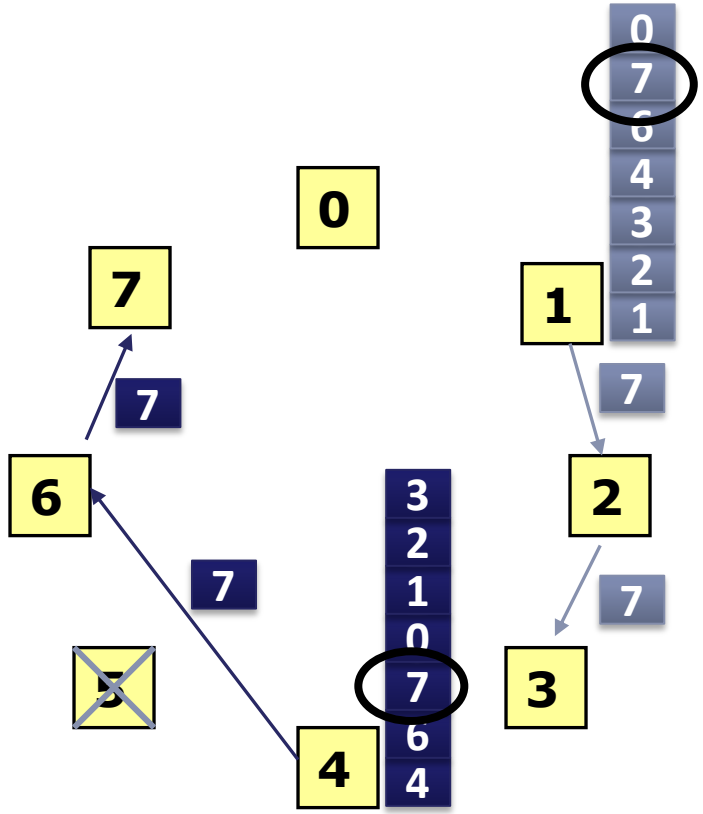
  - Se o sucessor tiver falhado então o processo envia a mensagem para os próximos sucessores até encontrar um em funcionamento
- Sempre que um processo **recebe** a mensagem **`ELEIÇÃO`**, **adiciona-lhe o seu identificador** (para poder ser candidato a líder) e passa-a ao seu sucessor (ou seguinte que esteja activo)
- Quando a **mensagem retorna a P** (que iniciou o algoritmo):
  - este escolhe **deterministicamente** o próximo coordenador (por exemplo, o que tiver o maior identificador),
  - e em seguida **faz circular** uma mensagem **`COORDENADOR`**

  `COORDENADOR = <ID do coordenador, lista de processos>`
- A mensagem é retirada do círculo quando volta ao processo que a criou (P)

Algoritmo baseado em anel: exemplo



Envio da mensagem **ELEIÇÃO** por dois processos distintos



Envio da mensagem **COORDENADOR** pelos dois processos (ambos escolhem o mesmo pois a escolha é determinística)

refletir.com

- Considere um grupo estático com 4 processos (1, 2, 3 e 4) em que apenas o processo 2 detecta a falha do processo 4. Descreva a execução do algoritmo “bully” na visão de cada processo do sistema.
  - 1
  - 2
  - 3
  - 4

Bibliografia recomendada

- [Coulouris et al]
  - Secções 15.1 a 15.3

- [van Steen, A.S. Tanenbaum]
  - Secções 6.3 e 6.4