UNIVERSIDADE FEDERAL DO CEARÁ - Campus Quixadá
Cursos: SI, ES, RC, CC e EC
Código: QXD0043
Disciplina: Sistemas Distribuídos

Capítulo 12 – Coordenação e Acordo

Prof. Rafael Braga

Agenda

- 12.2 – Exclusão mútua distribuída
  - Detecção de falhas
  - Algoritmos
- 12.3 – Eleições
- 12.4 – Comunicação *multicast*
- 12.5 – Consenso e problemas relacionados

© Pearson Education 2012

Detecção de falhas

- **Detector de falhas (DF)**: serviço que informa se um processo está vivo.
- Duas categorias de DF:
  - **Não confiáveis**
    - Respostas = {não suspeito, suspeito}
  - **Confiáveis**
    - Respostas = {não suspeito, « morto »}
    - Implementação possível…

© Pearson Education 2012

Detector de falhas: implementação

- A cada **T** segundos
  - Todo processo p envia uma msg « estou aqui » para todos os outros processos
- DF confiável
  - Sistemas síncronos
  - Nenhuma msg « estou aqui » em **T + A** segundos
  - A é conhecido
- DF não confiável
  - Sistemas assíncronos
  - Nenhuma msg « estou aqui » em **T + E** segundos
  - **E = timeout** deve ser estimado
  - Problema: como calibrar E ?

© Pearson Education 2012

Detector de falhas: implementação

- Como calibrar o valor de E ?
  - Usar timeouts que reflitam as condições de retardo atuais da rede
  - Se E for muito pequeno (ex. 0.1 seg)
    - Desperdício de *bandwidth*
    - Muitos processos serão suspeitos
  - Se E for muito grande (ex. 1 semana)
    - Processos mortos podem ser considerados « não suspeitos »

© Pearson Education 2012

Exclusão mútua

- **Problema**:
  - Quando um conjunto de processos compartilha um conjunto de recursos;
  - Como atribuir a um dos processos o direito de acessar um determinado recurso (arquivo, janela, hardware) ?
  - Idêntico ao problema de sessão crítica em SO's, mas solução deve ser baseada em troca de mensagens.
- **Exemplos**:
  - Acesso a arquivos: *file-locking service* UNIX (lockd daemon);
  - Acesso ao "meio compartilhado" em redes Ethernet, IEEE 802.11.

© Pearson Education 2012

Requisitos básicos

- EM1: (segurança): No máximo um processo por vez pode ser executado na seção crítica (SC) ou região crítica (RC).
- EM2: (subsistência): Os pedidos para entrar e sair da seção crítica têm sucesso.
  - Implica em independência de impasse e inanição;
- EM3: (ordenação): Se um pedido para entrar na SC aconteceu antes de outro, então a entrada na SC é garantida nessa ordem.

© Pearson Education 2012

Avaliação de desempenho

- *Largura de banda* consumida, que é proporcional ao número de mensagens envidas em cada operação de entrada e saída;
- *Atraso do cliente* acarretado por um processo em cada operação de entrada e saída;
- *Rendimento ou Throughput do sistema*, que trata-se da velocidade com que o conjunto de processos como um todo pode acessar a seção crítica.

© Pearson Education 2012

Exclusão Mútua Distribuída

- **Algoritmos**
  - Centralizado
  - Distribuído
  - Anel
  - *Multicast* e relógios lógicos [Ricart e Agrawala]
  - Votação de Maekawa

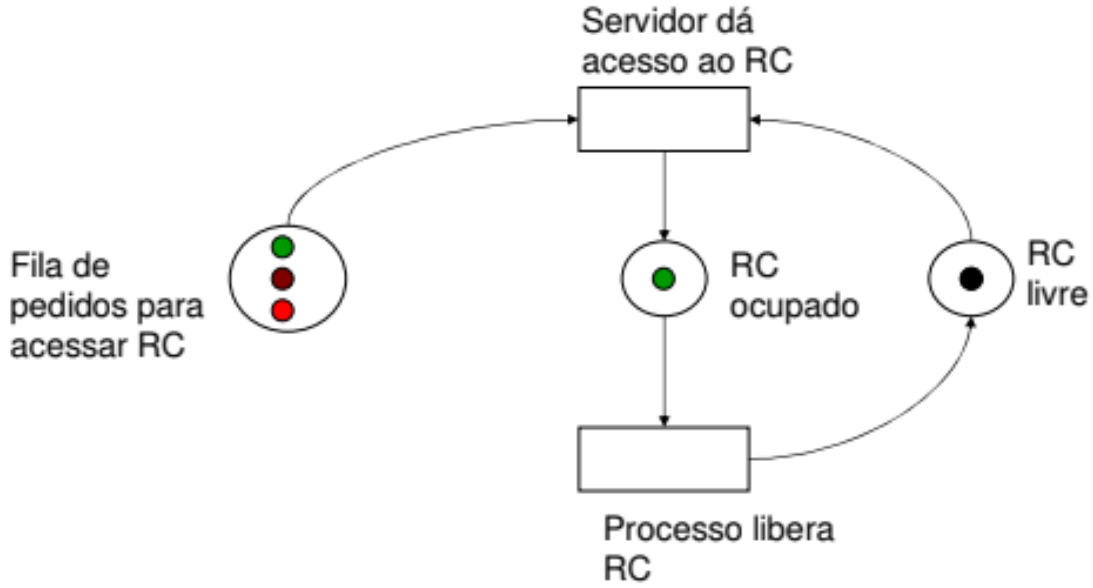
© Pearson Education 2012

Exclusão Mútua: Algoritmo Centralizado

a) Processo verde pede permissão ao coordenador para entrar em uma região crítica. A permissão é concedida.

b) Processo vermelho então pede permissão para entrar na mesma região crítica. O coordenador não responde.

c) Quando o processo verde sai da região crítica, ele informa ao coordenador, que então responde ao processo 2.



RC = recurso crítico

© Pearson Education 2012

Exclusão Mútua: Algoritmo Centralizado

- **Avaliação do algoritmo**
  - Assumindo que falhas não ocorrem, as seguintes propriedades são verificadas:
    - **Seguro**: somente um processo pode acessar o RC
    - **Vivo**: pedidos para ganhar e liberar o RC são atendidos
    - **Justo**: o acesso ao RC pode ocorrer numa ordem diferente dos pedidos (*HB = relação happened before*).

© Pearson Education 2012

Exclusão Mútua: Algoritmo Centralizado

- **Desempenho**
  - **Servidor** pode ser um gargalo
  - **Consumo de bandwitdh**:
    - Para acessar o RC: 2 mensagens (espera = 1 RTT)
      - pedido do proc. requerente (mesmo quando não há pedidos)
      - **1** autorização do servidor
    - Para liberar o RC:
      - **1** mensagem (espera = 0, assumindo com. assíncrona)
  - **Tempo de sincronização = 1 RTT**:
    - **1** mensagem de liberação do RC (processo -> servidor)
    - **1** mensagem de autorização de acesso ao RC (servidor -> processo)

© Pearson Education 2012

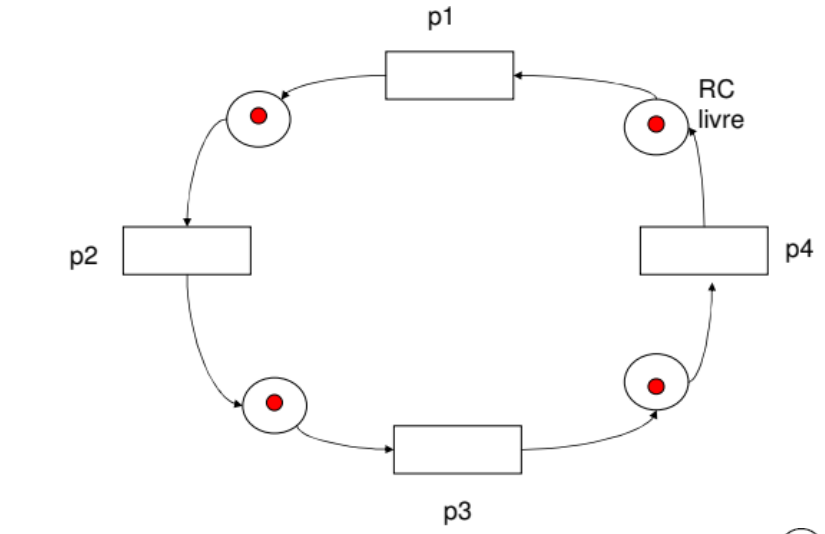
Exclusão Mútua: Algoritmo em Anel

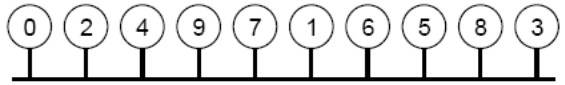
- Maneira simples de construir a exclusão mútua entre N processos, sem exigir um processo adicional.

a) Um grupo não-ordenado de processos em uma rede.

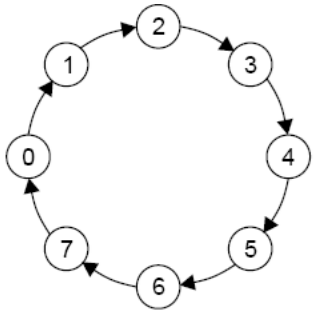
b) Um anel lógico construído em software.

c) Token com permissão para acesso à RC circula pelo anel





(a)



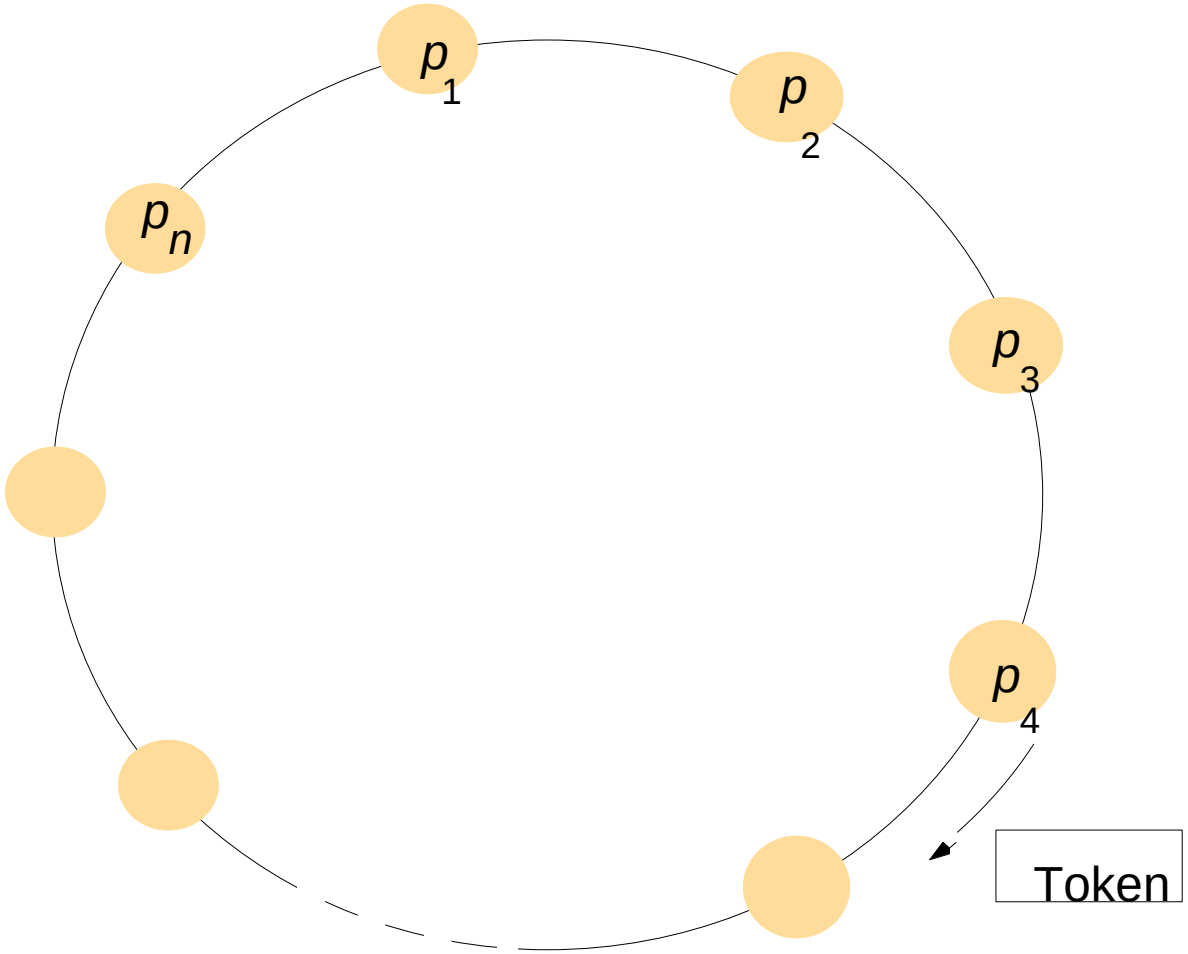
(b)

RC = RECURSO CRÍTICO

© Pearson Education 2012

Exclusão Mútua: Algoritmo em Anel

- Token de exclusão mútua circulando no anel;
- Se um processo não pede pra entrar na seção crítica ao receber o *token*, então ele encaminha imediatamente o *token* para o seu vizinho.



© Pearson Education 2012

Exclusão Mútua: Algoritmo em Anel

- **Avaliação**
  - Assumindo que falhas não ocorrem, as seguintes propriedades são verificadas:
    - **Seguro**: somente um processo pode acessar o RC
    - **Vivo**: pedidos para ACESSAR e LIBERAR o RC são atendidos
    - **Justo**: o acesso ao RC pode ocorrer numa ordem diferente dos pedidos

© Pearson Education 2012

Exclusão Mútua: Algoritmo em Anel

- **Avaliação de Desempenho**
  - Consumo de bandwitdh:
    - Contínuo, salvo quando um dos processos utiliza o RC
    - Acesso ao RC pode variar entre **0 e N** mensagens
      - **Zero** quando acabou de receber o token
      - **N** quando acabou de passar o token
    - Liberação do RC requer 1 (**UMA)** mensagem
  - Tempo de sincronização:
    - Entre **1** e **N** transmissões de mensagens

© Pearson Education 2012

Exclusão mútua: Ricart e Agrawala

- Baseado em ***multicast*** **e relógios lógicos de LAMPORT**
- **Ideia geral**: um processo só ganha o RC quando todos os outros processos aceitarem seu pedido
- **Cada processo**:
  - Atualiza seu **relógio de Lamport** inclusive em todas as mensagens de pedido de acesso ao RC
  - Se comporta como um **máquina de estados finitos**:
    - **{RC livre, RC solicitado, RC em uso}**

© Pearson Education 2012

Exclusão mútua: Ricart e Agrawala

- **Esboço do algoritmo**
  - ***Se*** no **« broadcast »** de um pedido os estados de todos os outros processos = **« RC LIVRE »**
    então o requerente pode acessar o RC
  - ***Se*** pelo menos um dos processos o utiliza: estado **« RC EM USO »**
    então este processo não **responderá** até liberar o RC
  - ***Se*** dois ou mais processos solicitam acesso ao mesmo tempo
    então aquele que tem o pedido com o menor *timestamp* ganhará a autorização
  - ***Se*** os *timestamps* forem iguais
    então o processo com o menor ID ganha a autorização

© Pearson Education 2012

Exclusão mútua: Ricart e Agrawala

Inicialização

- *estado* := RC LIVRE;

Para ganhar acesso ao RC

- *estado* := RC SOLICITADO;
- **Faça** o multicast do pedido para todos processos;
- ***T*** := timestamp do pedido;

} request processing deferred here

- ***Wait*** *até que o nº de respostas recebidas* seja = $(N - 1)$;
- *estado* := RC EM USO;

P_j recebe um pedido $<T_i, p_i>$ de p_i

- ***if*** (*state* = C EM USO or (*estado* = RC SOLICITADO *and* $(T, p_j) < (T_i, p_i)$))
- ***then***
- coloque na fila o *pedido de* p_i sem responder;
- ***else***
- responda imediatamente para p_i;
- **end if**

Para liberar o RC

- *estado* := RC LIVRE;
- **responda** para todos os pedidos enfileirados (linha 3 do item anterior);

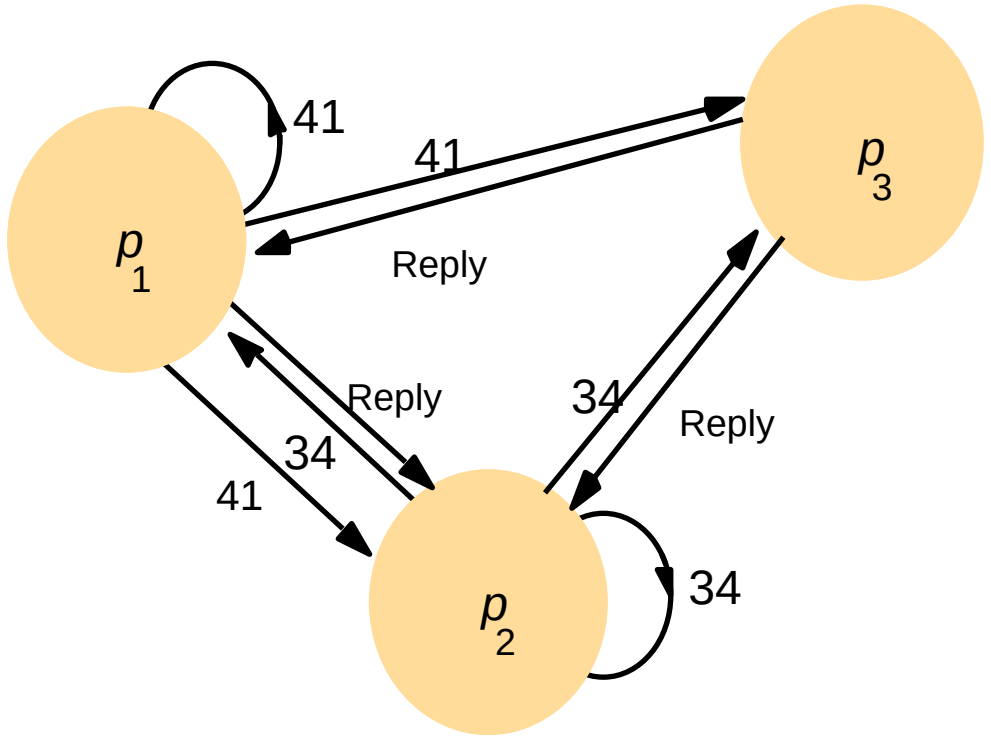
© Pearson Education 2012

Exclusão mútua: Ricart e Agrawala

- Exemplo

Tempo físico

P1	P2	P3
Request **RC SOLICITADO** t=41	Request **RC SOLICITADO** t=34	**RC SOLICITADO**
P1 recebe (34, p2) e responde	P2 recebe (41, p1) e põe na fila pois (34, p2) < (41, p1)	P3 recebe (41, p1) e responde, P3 recebe (34, p2) e responde
	P2 acessa o RC **RC EM USO**	
	P2 libera o RC e envia resposta para todos os processos da fila = p1 **RC LIVRE**	



© Pearson Education 2012

Exclusão mútua: Ricart e Agrawala

- **Avaliação**
  - Assumindo que falhas não ocorrem, as seguintes propriedades são verificadas:
    - **Seguro**: somente um processo pode acessar o recurso crítico
    - **Vivo**: pedidos para acessar e liberar o RC são atendidos
    - **Justo**: o acesso ao RC ocorre na mesma ordem dos pedidos

© Pearson Education 2012

Exclusão mútua: Ricart e Agrawala

- **Desempenho**
  - Consumo de *bandwitdh*:
    - Se não existir HW para *multicast* então para ACESSAR O RC: **2 * (N – 1)** msgs
      - isto é, **(N – 1)** para fazer o *multicast* do pedido mais **(N – 1)** respostas
    - Se existe HW para *multicast* então temos um consumo = **N** mensagens
      - isto é, **1** para fazer o *multicast* do pedido mais **(N-1)** respostas
  - Tempo de sincronização:
    - **Tempo de transmissão de uma mensagem (melhor caso)**
      Quando um processo utiliza o RC e um outro o solicita basta uma msg de liberação do utilizador para o requerente

© Pearson Education 2012

Exclusão mútua: alg. de votação de Maekawa

- Processos não precisam da autorização de todos para acessar o RC → somente de um subconjunto
- Como dividir os processos em subconjuntos ?
  - Para todo p_i (i = 1, 2, …, N)
    - Associa-se um conjunto votante (ou de eleitores) v_i tal que
    - $p_i \in v_i$
    - $v_i \cap v_j \neq \emptyset$ deve haver pelo menos um membro comum dados dois subconjuntos votantes
    - $| v_i |$ = K todos os subconjuntos votantes tem mesmo tamanho
    - Cada processo p_j está contido em M subconjuntos votantes

© Pearson Education 2012

Exclusão mútua: alg. de votação de Maekawa

1. Para acessar o RC, **pi envia** pedido para os **K** membros de Vi (inclusive para ele mesmo)
2. Não pode entrar até receber as **K** respostas
3. Quando libera o RC, **envia** mensagens liberação para todos membros de Vi (inclusive para ele mesmo)
4. Quando um **pj** recebe um pedido

   Se estado **=** « RC em uso » ou já votou

   Coloca o pedido de pi na fila sem responder

   Senão

   Responde imediatamente (vota)
5. Quando **pj** recebe uma msg de liberação do RC de **pi**
   Remove a **cabeça** da fila e envia uma resposta

© Pearson Education 2012

Exclusão mútua: alg. de votação de Maekawa

- **Inicialização**
  - estado := RC-LIVRE
  - votou := F
- **Para p_i entrar na SC**
  - estado := RC-SOLICITADO
  - Multicast do pedido para todos proc. em V_i (inclusive para p_i)
  - Wait until (n° respostas recebidas = K)
  - estado := RC-EM-USO
- **Qdo p_j recebe um pedido de p_i**
  - Se (estado$_j$=RC-EM-USO ou votou$_j$ =V)
    - Colocar pedido de p_i na fila sem respondê-lo
  - Senão
    - Enviar resposta para p_i
    - votou := V
- **Para p_i liberar o RC**
  - estado := RC-LIVRE
  - Multicast da msg de liberação para todos os proc. em V_i (inclusive para ele mesmo)
- **Qdo p_j recebe uma msg de liberação de p_i**
  - Se (fila de pedidos está vazia)
    - Votou := F
  - Senão
    - Remova a cabeça (digamos p_k)
    - Envia resposta para p_k
    - Votou := V

© Pearson Education 2012

Exclusão mútua: alg. de votação de Maekawa

- **Avaliação**
  - Assumindo que falhas não ocorrem, as propriedades são verificadas:
    - **Seguro**: **sim,** somente um processo pode estar utilizar o RC
    - **Vivo**: **não**, algoritmo sujeito a deadlocks
    - **Justo: sim,** se o algoritmo for alterado para não entrar em deadlock usando relógios lógicos
- **Desempenho**
  - Consumo de *bandwitdh*:
    - Para acessar o RC: $2\sqrt{N}$ (N é o número de processos)
    - Para liberar o RC: $\sqrt{N}$ (sem *hardware* para *multicast*)
  - Tempo de sincronização:
    - **1 RTT**

© Pearson Education 2012

Exclusão mútua: conclusão

- Nenhum dos algoritmos tolera incondicionalmente a perda de mensagens
  - O **algoritmo do servidor central** tolera falhas de um processo cliente desde que ele não tenha solicitado a SC nem esteja na SC
  - O **algoritmo em anel** não tolera falhas de nenhum dos processos
  - O **algoritmo de Ricart e Agrawala** pode ser modificado para ser tolerante a falhas: se o processo falhou, assume-se implicitamente que ele enviou uma resposta para o requerente
  - O **algoritmo de Maekawa** pode tolerar falhas desde que o processo que falhou não participe do conjunto de eleitores atualmente solicitado

© Pearson Education 2012

Exclusão mútua: conclusão

	Servidor Central	Anel	Ricart e Agrawala	Maekawa
Entrar na SC	2	[0, N]	2*(N-1)	$2*N^{1/2}$
Sair da SC	1	1	[0, N-1]	$N^{1/2}$
Consumo total de bandwidth	3	[1, N+1]	[2*(N-1), 3*(N-1)]	$3*N^{1/2}$
Delay de sincronização	2	[1,N]	1	2

© Pearson Education 2012

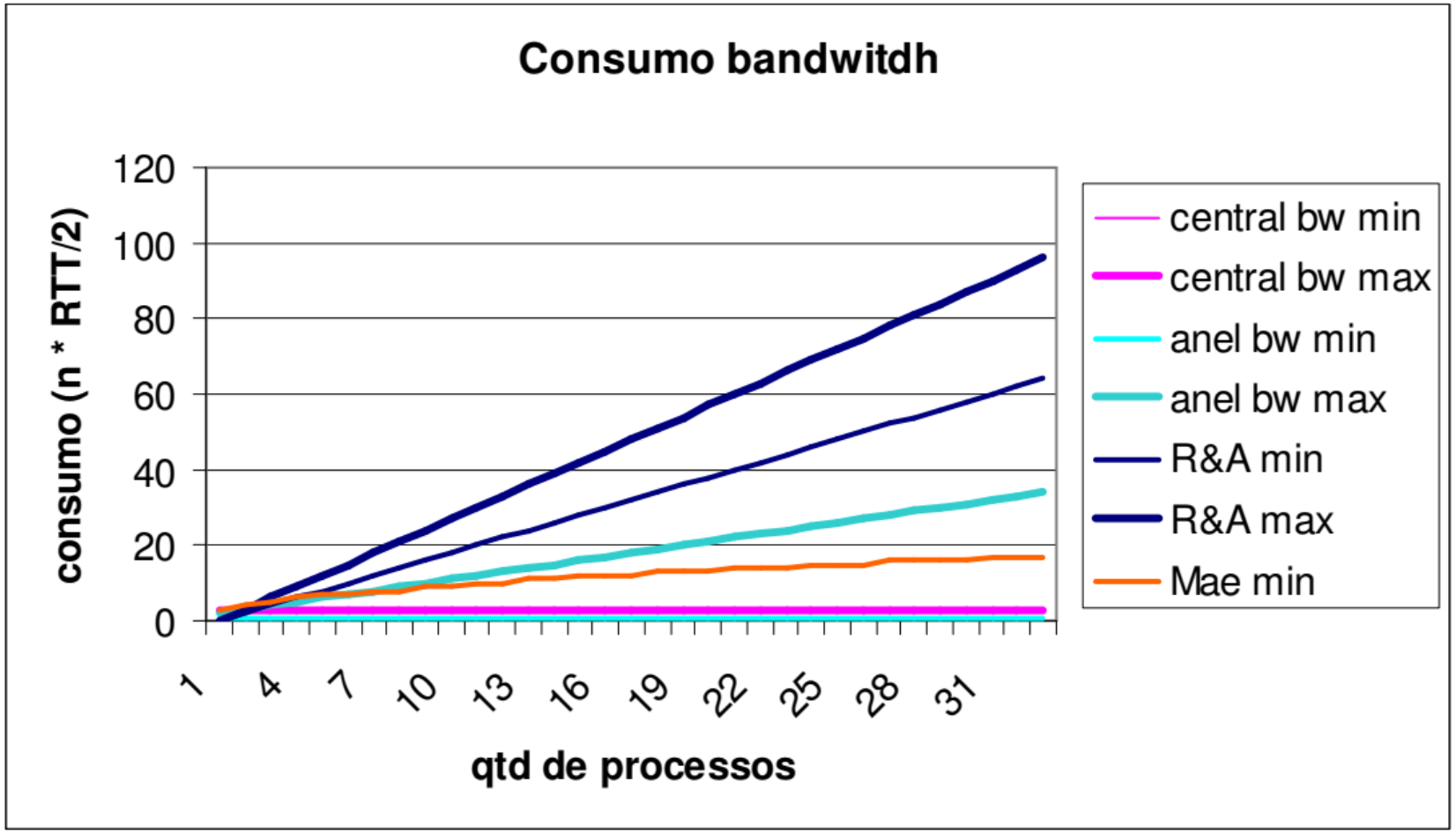
Comparação

Uma comparação de três algoritmos de exclusão mútua.

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

© Pearson Education 2012

Exclusão mútua: conclusão



© Pearson Education 2012

Agenda

- 12.2 – Exclusão mútua distribuída
- 12.3 – Eleições
- 12.4 – Comunicação *multicast*
- 12.5 – Consenso e problemas relacionados

© Pearson Education 2012

Algoritmos de Eleição Distribuída

- Algoritmos em anel
- Valentão (*Bully*)

© Pearson Education 2012

Terminologia e hipóteses

- **Objetivo:** eleger um coordenador (aplicações: exclusão mútua com controle centralizado, detecção de deadlock distribuído)
- Dizemos que um processo **convoca eleições** quando ele inicia um algoritmo de eleição.
- Um processo **não convoca mais de uma** eleição simultaneamente
- Porém, **N processos** podem convocar **N eleições** concorrentemente
- **O ganhador deve ser único** mesmo se dois processos convocaram eleições concorrentemente
- Um processo é **participante ou não participante** de uma eleição

© Pearson Education 2012

Terminologia e hipóteses

- Um processo será eleito em função de um **identificador**.
- Um **identificador** é qualquer **valor útil** (ex. tempo de resposta)
- **Identificadores** devem ser **únicos** e **ordenados** → quando são iguais para dois processos saberemos qual é o ganhador
- Aquele com o **maior identificador** é o **eleito**.
- Eleger o processo com **menor tempo de resposta**
  - P2 = <1/25 ms, 2> = <valor, id do processo**>** = **identificador**
  - P3 = <1/25 ms, 3>
  - **P1 = <1/20 ms, 1>**

© Pearson Education 2012

Terminologia e hipóteses

- Cada processo p_i possui uma variável $eleito_i$
- A variável $eleito_i$ contém o **identificador** do processo eleito
- Se $eleito_i = ND$ significa que o eleito ainda não foi definido

© Pearson Education 2012

Terminologia e hipóteses

- **Propriedades desejáveis** para os algoritmos:
  - **E1 (seguro)**
    - para toda execução, todo processo participante p_i terá $eleito_i$ = ND ou eleito i=P onde P é um processo em execução (não abortou) e com o maior identificador (*todos os processos devem ter o mesmo vencedor ao final*)
  - **E2 (vivacidade):** todos processos p_i participam e eventualmente atribuem um valor diferente de *ND* para $eleito_i$

© Pearson Education 2012

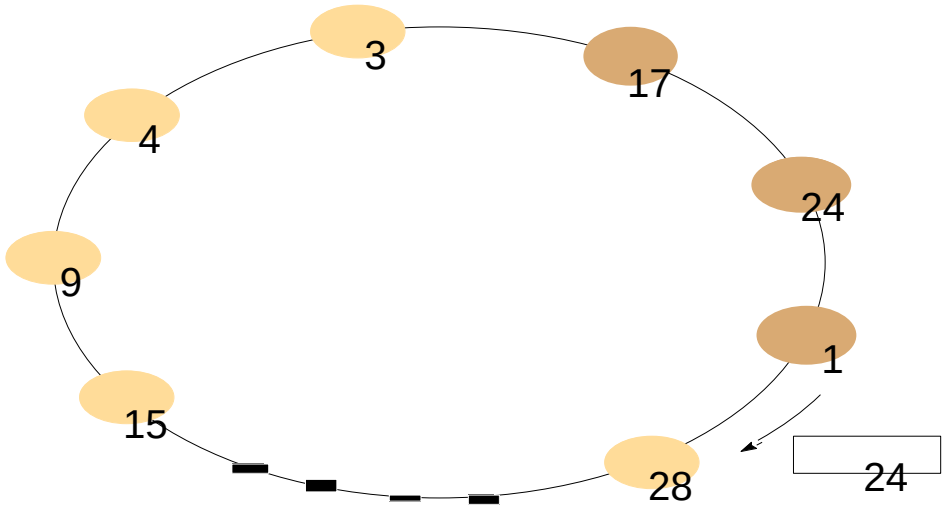
Terminologia e hipóteses

- **Desempenho**
  - **Bandwidth**: proporcional à quando de mensagens enviadas
  - **Turnaround**: o número de tempos de transmissão das mensagens ao longo do tempo
    - msgs enviadas em paralelo contam como um tempo de transmissão

© Pearson Education 2012

Eleição baseada em Anel (Chang e Roberts, 79)

- **Objetivo**: eleger um processo com o maior identificador!
- Identificadores são **únicos**
- Assume-se: **falhas** não ocorrem/sistemas assíncrono
- Processos são organizados num **anel lógico**
- Mensagens são enviadas no **sentido horário**
- **Funcionamento**
  - **2 tipos de mensagem**: de eleição e do eleito



- A eleição foi iniciada pelo processo 17.
- Até o momento o maior identificador de processo encontrado é 24.
  - Processos que já estão participando da eleição são mostrados em cor escura

© Pearson Education 2012

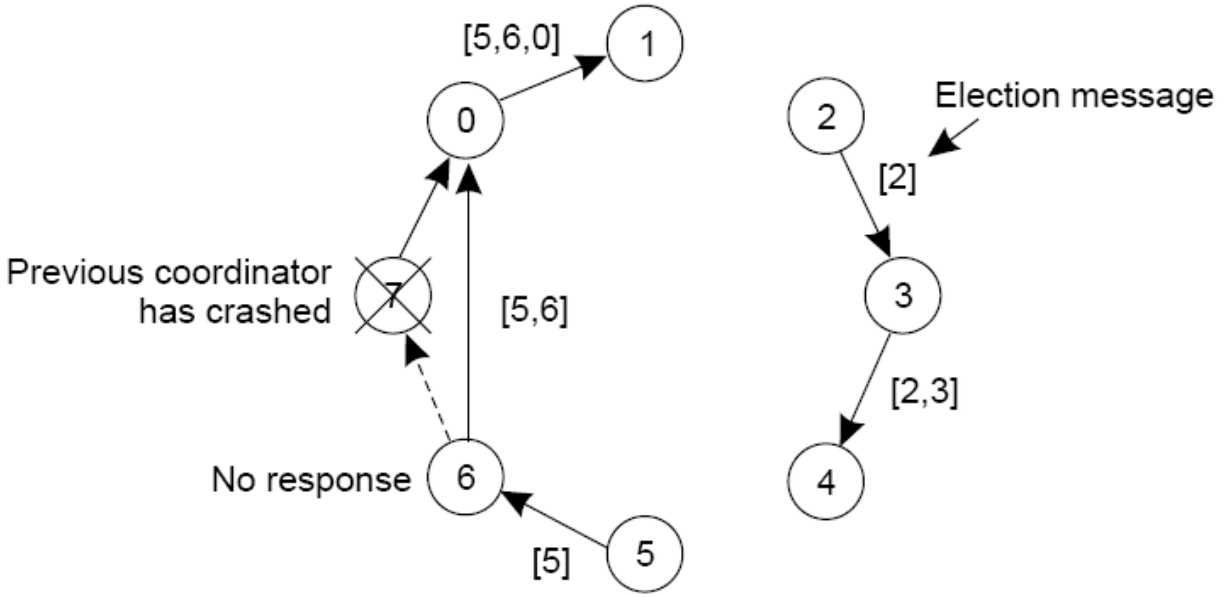
Eleição baseada em Anel (Chang e Roberts, 79)

- **Inicialmente** todos os processos são *não participantes*
- O **iniciador** marca-se como *participante* e envia seu identificador, i.e. uma **msg de eleição**, ao vizinho
- Ao receber uma **msg de eleição**
  - Se não participante
    - Se **id local < id recebido**
      - Forward id recebido
    - Senão
      - Forward id local
    - Marcar-se como participante
  - Se participante
    - Se **id local > id recebido**
      - Não envia msg de eleição
    - Se **id local < id recebido**
      - Forward msg de eleição
    - Se **id local = id recebido**
      - Marcar-se como coordenador
      - Marcar-se como não participante
      - Enviar **msg de eleito** (com o id do proc) para o vizinho
- Ao receber uma **msg de eleito**
  - Marcar-se como não participante
  - *Eleito i* := id do eleito
  - Se não for o coordenador
    - Forward msg para o vizinho

© Pearson Education 2012

O Algoritmo do Anel

- Eleição utilizando um anel lógico de processos.
- Neste exemplo, há um particionamento da rede e dois líderes são eleitos!



© Pearson Education 2012

Algoritmo valentão (*Bully)*

- Processos podem « **falhar** » durante a execução
- **Utilizável em sistemas síncronos**: usa timeout para detectar falhas
- Assume que:
  - um **processo conhece** todos os que tem identificadores maiores e que este processo pode comunicar-se com todos os outros
- **3 tipos** de mensagens:
  - **de eleição**: anuncia uma eleição
  - **resposta**: resposta a uma msg de eleição
  - **coordenação**: anuncia o processo eleito
- Qualquer processo pode iniciar uma eleição quando detecta que o coordenador morreu!
- Várias eleições podem ser iniciadas concorrentemente

© Pearson Education 2012

Algoritmo valentão (*Bully)*

- **Algoritmo inicia** quando um dos processos detecta que o coordenador falhou.
- **Se o iniciador tem o maior identificador então**
  - elege-se como coordenador enviando uma msg coordenadora para os processos com identificadores menores
- **senão**
  - envia uma msg de eleição para aqueles com identificadores maiores
  - **Aguarda** uma « resposta » no intervalo **T**
  - **SE** receber UMA resposta dentro de **T então**
    - Aguarda uma **msg de coordenação** dentro do intervalo **T'**
    - Se não receber, inicia nova eleição
  - **Senão**
    - o processo **elege-se** enviando uma **msg de coorenação** para todos os processos com identficadores menores

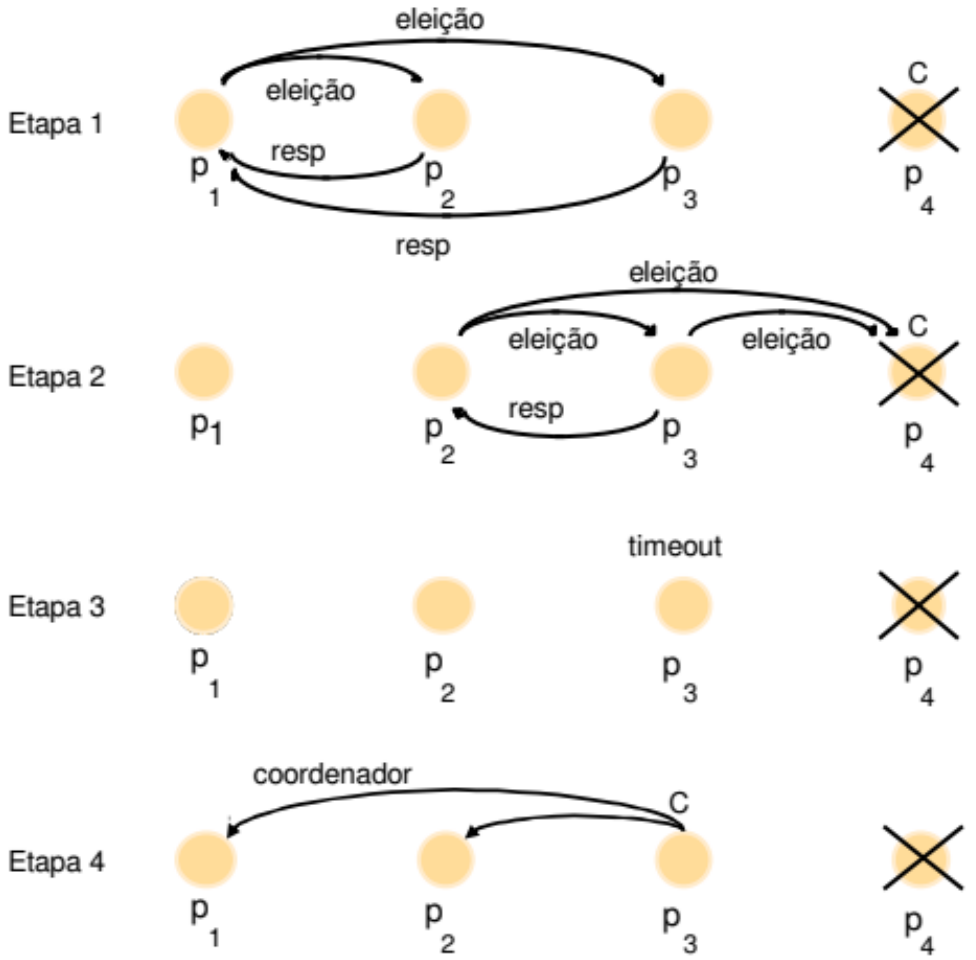
- **Se** um p_i **recebe msg de coordenação então** $eleito_i$:= id do coordenador
- **Se** um p_i **recebe msg de eleição então**
  - Se p_i não iniciou nenhuma eleição então
    - p_i retorna uma resposta
    - p_i inicia nova eleição
- Qdo um processo substitui um outro que falhou ele inicia uma eleição.
  - Se este novo processo tem o maior identificador então elege-se coordenador mesmo que o coordenador atual esteja funcionando, daí o nome **V A L E N T Ã O**

- **OBS**: como o sistema é **síncrono** pode se construir um detector de falhas confiável. **T** = $2T_{trans} + T_{proc}$

© Pearson Education 2012

Algoritmo de Bullying: exemplo 1

- **Exemplo**
  - p1 detecta falha
  - Somente p4 falha



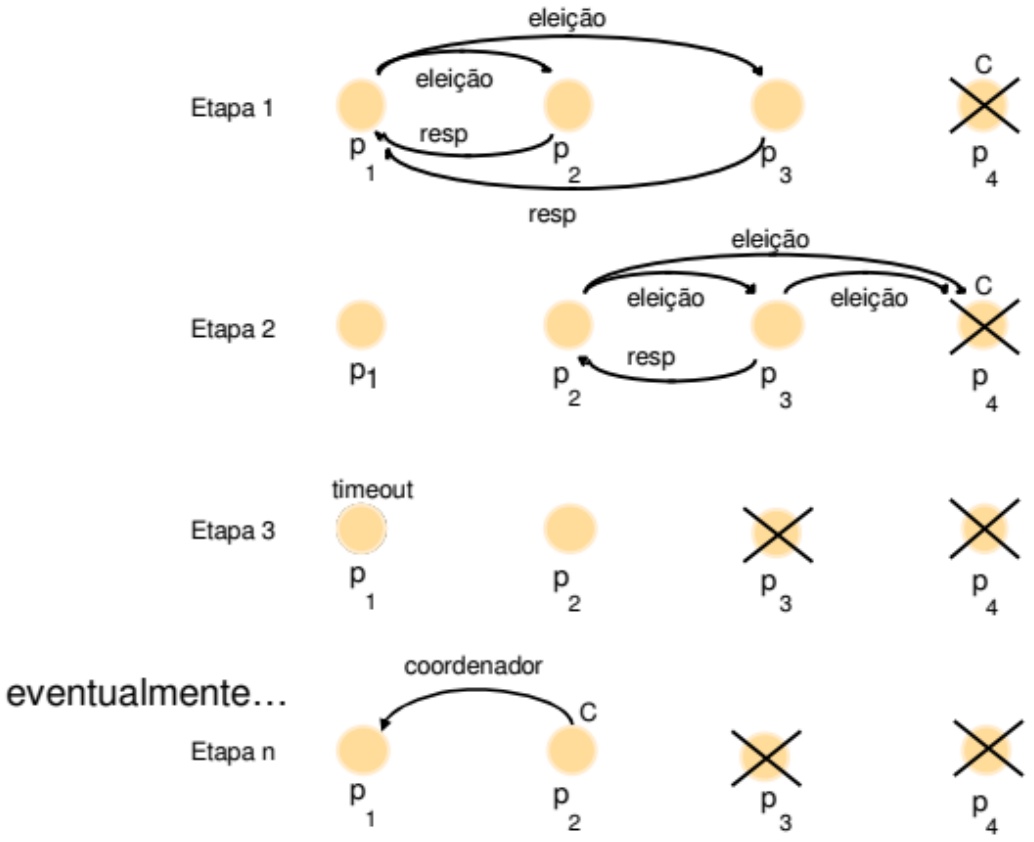
© Pearson Education 2012

Algoritmo de Bullying: exemplo 1

- P_1 detecta falha de p_4 (inicio do algoritmo)
- Eleição do coordenador p_2,

após a falha de p_4 e, em seguida, de p_3

- Falha inicialmente detectada por p_1

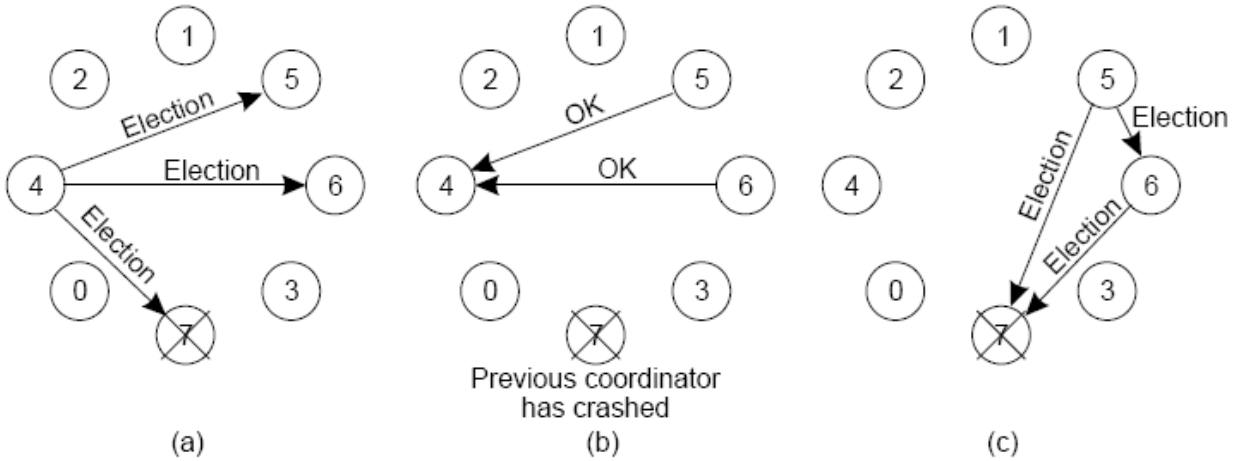


© Pearson Education 2012

O Algoritmo de "*Bullying*": exemplo 2

O algoritmo de eleição de *bullying*

a) Processo 4 inicia a eleição (após detectar a falha do antigo líder)
b) Processos 5 e 6 respondem, dizendo ao processo 4 para parar
c) Agora os processos 5 e 6 cada um iniciam suas eleições



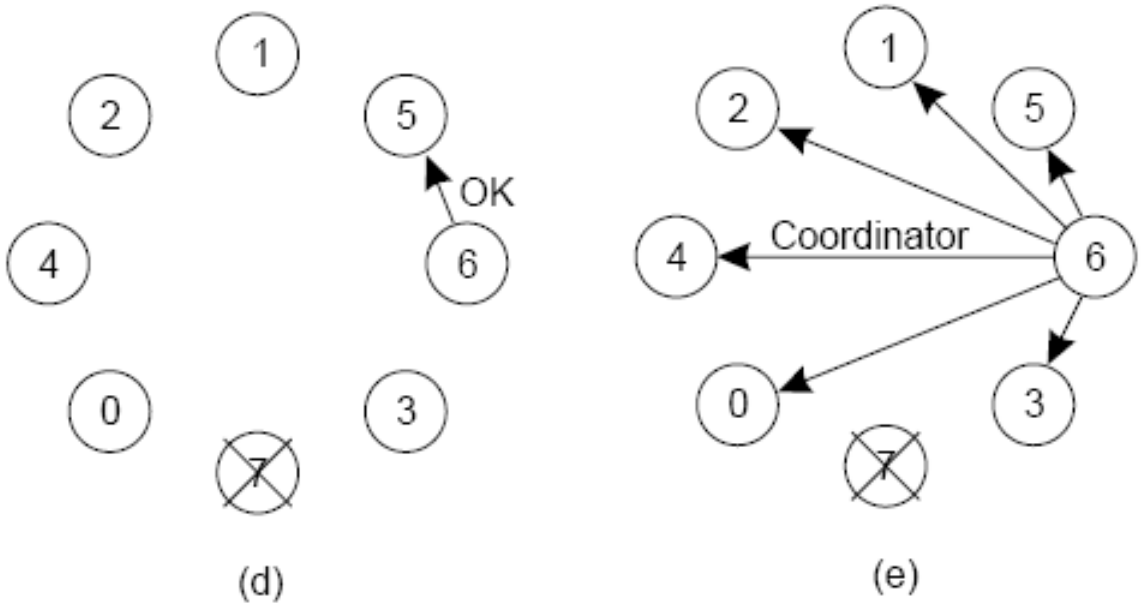
Suposição: todo processo conhece os processos "superiores" a ele

© Pearson Education 2012

O Algoritmo de "*Bullying*": exemplo 2

d) Processo 6 diz ao processo 5 para parar

e) Processo 6 vence a eleição e avisa a todos os demais



O que acontece na "volta" do processo 7?

© Pearson Education 2012

Algoritmo valentão (*Bully)*

Avaliação:

- **Respeita os requisitos ?**
  - **Não é seguro pois:**
    - um processo substituto de um processo **p** pode decidir que ele tem o maior identificador ao mesmo tempo que aquele que detectou a falha de p. Os dois declaram-se coordenadores e os outros processos podem assumir ou um ou outro como tal (não há garantia na ordem de entrega das mensagens).
  - **É vivo:** assumindo-se entrega confiável das mensagens

© Pearson Education 2012

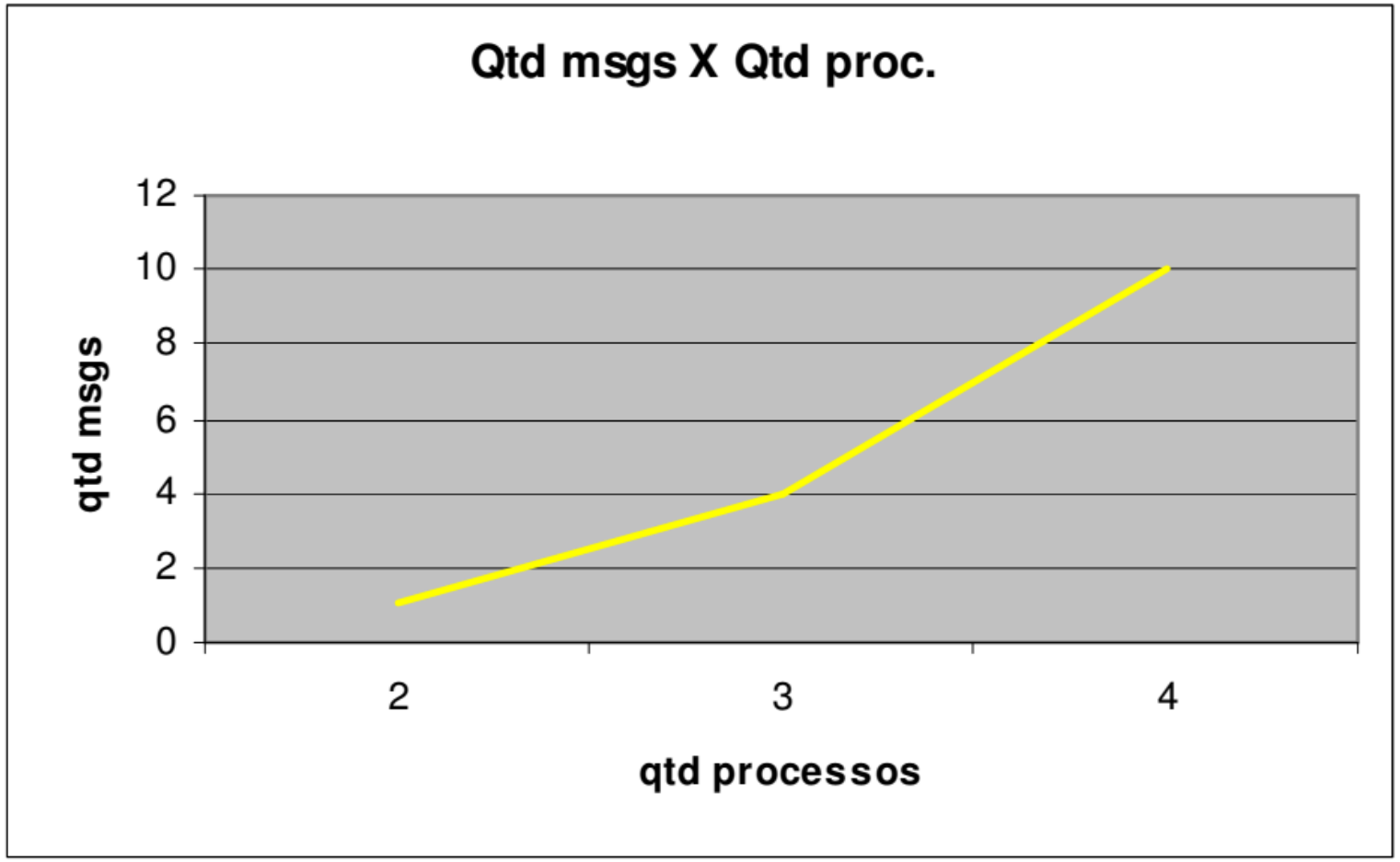
Algoritmo valentão (*Bully)*

Avaliação:

- **Desempenho**
  - **Melhor caso**:
    - **Bandwidth**: quando o proc. com o segundo maior identificador detecta a falha do coordenador elege-se com **N – 2** mensagens coordenadoras
    - **Turnaround**: **1 =** *o número de tempos de transmissão das mensagens (serializados)*
  - **Pior caso**:
    - **Bandwidth**: quando o processo com o menor identificador detecta a falha do coordenador → O(N2) *função do número de processos*
    - **Turnaround**: *idem*

© Pearson Education 2012

Algoritmo valentão (*Bully)*



© Pearson Education 2012

Agenda

- 12.2 – Exclusão mútua distribuída
- 12.3 – Eleições
- 12.4 – Comunicação *multicast*
- 12.5 – Consenso e problemas relacionados

© Pearson Education 2012

O Consenso Distribuído

O Consenso é um problema fundamental em Sistemas Distribuídos pois é:

- utilizado como módulo fundamental de vários algoritmos onde os processos precisam ter uma visão (ou ação) idêntica. Exemplos:
  - para ordenação total de eventos (ou mensagens);
  - sobre o conjunto de processos não falhos/ativos de um grupo;
  - colaboração entre agentes em sistemas multiagentes;
- pode-se mostrar a sua equivalência com outros problemas tais como:
  - *multicast* atômico (confiável + ordem total);
  - consistência interativa;
  - acordo bizantino.

Modelo do Sistema

- Conjunto de processos Pi (i= 1,2,..,N)
- A comunicação (por envio de mensagem) é confiável
- Os processos podem apresentar falhas, tipos:
  - falha de parada (crash)
  - falta arbitrária (bizantina)
- Até f processos podem falhar simultaneamente;
- Em alguns casos assumiremos que mensagens recebem uma assinatura digital para garantir a autenticidade do remetente;
- Impede-se que processos faltosos possam falsificar identidade do remetente.

O Consenso em sistema síncrono

- Definição do problema:
  - existem N processos, dos quais f processos apresentam falhas.
  - cada processo Pi propõe um único valor $vi \in D$
  - todos os processos interagem para a troca de valores entre sí
  - em algum momento, os processos entram no estado “decided” em que atribuem um valor para a variável de decisão *di (que não é mais alterada)*
  - Obs: O valor de *di* é uma função dos valores *vi* fornecidos pelos N-f processos corretos.

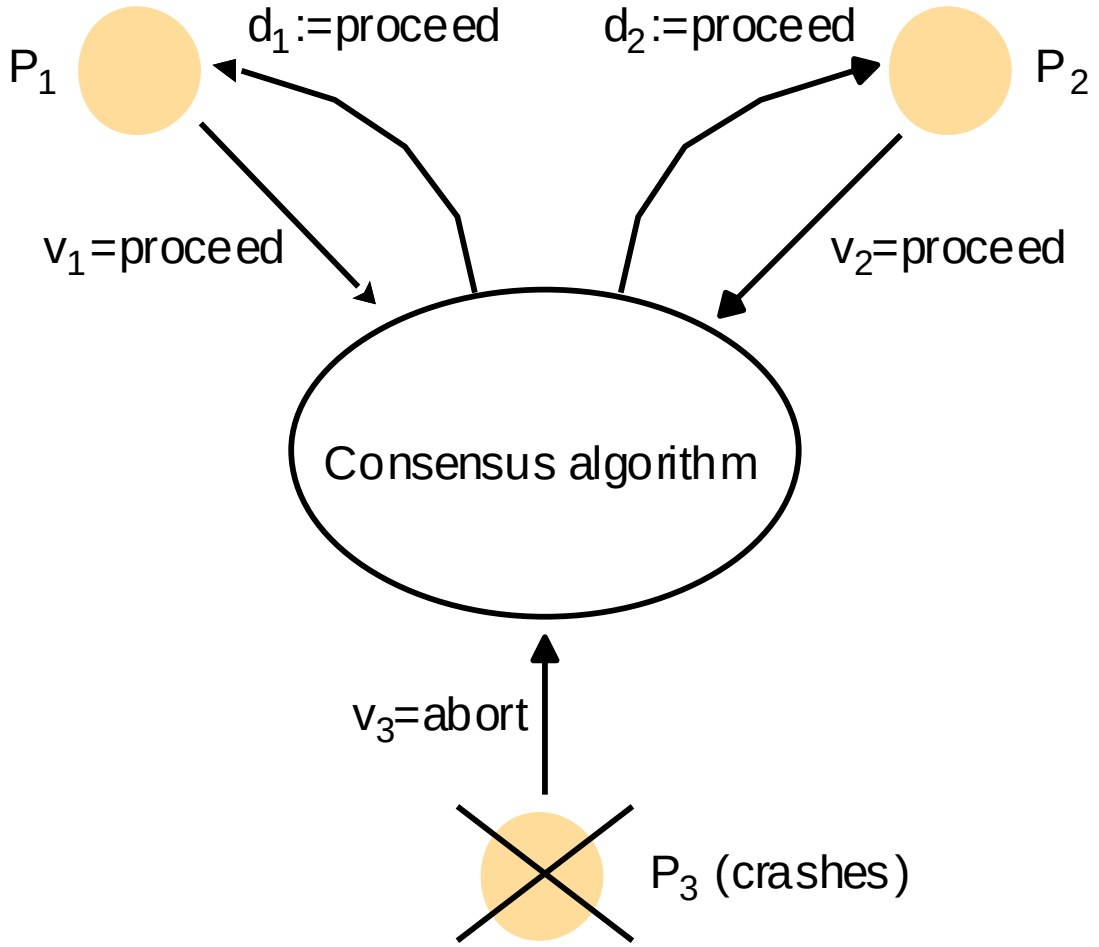
Principais Requisitos

- Terminação:
  - Em algum momento, cada processo correto atinge o estado "*decided*" e atribui um valor à variável de decisão di
- Acordo:
  - todos os processos corretos atribuem o mesmo valor para a variável de decisão
- Integridade
  - se todos os processos corretos propuseram o mesmo valor vi =v, então qualquer processo correto em "*decided*" também terá decidido di =v
- Integridade (alternativa mais fraca – depende da aplicação)
  - o valor de di (i=1,2,..,N) deve ser necessariamente igual ao valor proposto por um Pi correto

Exemplo

Consenso para 3 processos:

- P3 propõe NOK (*abort*), mas falha durante o consenso
- Os processos corretos decidem por OK (*proceed*)



© Pearson Education 2012

Um Algoritmo simples

- Assuma um grupo de processos corretos não faltosos. Cada processo *Pi* :
  - usa *multicast* confiável para mandar o valor vi para os demais
  - espera até receber N-1 mensagens
  - define o valor de *di* usando uma função determinística:
    - maioria(v1,v2,..,vN), ou ⊥ se não houver maioria,
    - máximo(v1,v2,..,vN) ou mínimo(v1,v2,..,vN)
    - média (v1,v2,..,vN)
- Terminação é garantida pelo *multicast* confiável
- Acordo e integridade garantidos pelo uso da mesma função (determinística) e do *multicast* confiável
- Mas e se processos podem falhar (crash ou bizantinos)?

O Consenso com falhas

- Se processos podem ter falhas tipo crash (omissão), então a terminação não estará garantida, a menos que se detecte a falha.
- Se o sistema é assíncrono pode ser impossível distinguir um *crash* de uma mensagem que demora um tempo indeterminado.
  - Fischer, Lynch & Paterson (1985) apresentaram um resultado teórico fundamental → é impossível garantir a obtenção de consenso em um sistema assíncrono com falhas tipo *crash*
- Se processos podem apresentar falhas arbitrárias (bizantinas), então processos falhos podem comunicar valores aleatórios aos demais processos, evitando que os corretos tenham a mesma base de dados $\{v_1, v_2, .., v_N\}$, para a tomada de uma decisão uniforme.

Consenso em sistemas síncronos

- [Dolev & Strong 1983] propuseram um algoritmo para o consenso entre N processos que tolera até k falhas tipo "*crash*" em um sistema síncrono.
- Ideia central: O algoritmo executa f+1 rodadas: Na rodada j todos os processos corretos difundem para todos os demais processos o conjunto de novos valores (propostos) v obtidos na rodada j-1.
- Inicialmente, difundem o próprio valor proposto. Ao final, executam uma função idêntica (p.ex. min/max) sobre o conjunto de valores coletados.
- Observações:
  - processos usam multicast não-confiável, mas sabe-se que a duração máxima de cada difusão é Δ;
  - a variável *values j* contém o conjunto de valores obtidos na rodada j;
  - os *timers* tem identificadores (que identificam a rodada corrente)

Consenso em sistemas síncronos

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \leq r \leq f + 1)$

B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

© Pearson Education 2012

Consenso em sistemas síncronos

- Corretude do Algoritmo de Dolev & Strong (1983):
  - *Terminação*: obvio, pois algoritmo termina após Δ*(f+1)
  - *Acordo e Integridade*: decorrem do uso da função min e se for mostrado que ao terminar, todos os processos têm conjuntos Values f+1 idênticos;
- Dolev & Strong (1983) mostraram que qualquer algoritmo para obter o consenso na presença de *f* falhas requer pelo menos *f*+1 rodadas.
- Este limite inferior também vale para falhas arbitrárias (Problema do Acordo Bizantino)

Problema dos Generais Bizantinos (PGB)

- O PGB foi inicialmente proposto por L. Lamport (1982) e equivale ao problema de consenso em um sistema com falhas arbitrárias.
- Motivação:
  - N generais (dentre eles, 1 comandante) devem concordar sobre *esperar* ou *atacar* uma cidade sitiada, que só conseguem conquistar se todos os batalhões atacarem conjuntamente.
  - O comandante dá a ordem (atacar/esperar), e os generais devem ter certeza que receberam o mesmo.
  - Alguns dos generais (ou o comandante) são traidores e tentam atrapalhar o consenso, divulgando para uns que receberam a ordem de *atacar*, e para outros a ordem de *esperar*.
- A principal diferença entre o PGB e consenso: um dos processos fornece um valor que deve ser acordado entre todos os processos corretos (ao invés de um valor proposto por cada processo, como no consenso)

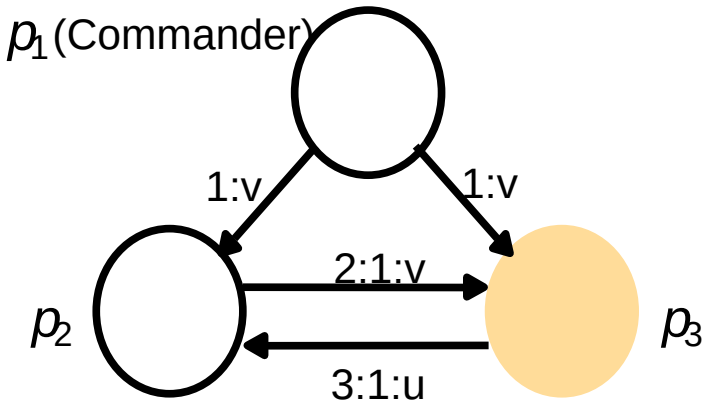
O Problema dos generais bizantino

- Os requisitos:
  - Terminação: Em algum momento, cada processo entra no estado “decidido” e atribui um valor à sua variável de decisão di
  - Acordo: Se pi e pj são corretos e estão no estado “decidido”, então di = dj
  - Integridade: Se o comandante está correto, então todos os processos usam como valor de di o valor proposto pelo comandante.
- Suposições:
  - Possibilidade de falhas arbitrárias = envio de qualquer msg a qualquer momento, ou omissão de envio
  - Até f processos (de N) podem falhar
  - Processos corretos conseguem detectar ausência de mensagem (usando timeouts), e neste caso assumem recebimento de um valor default → mas são incapazes de detectar falha do processo
  - Canais de comunicação são seguros: não é possível interceptar e modificar mensagens em trânsito
- **Ideia de solução**: todos contam para todos o que receberam do comandante

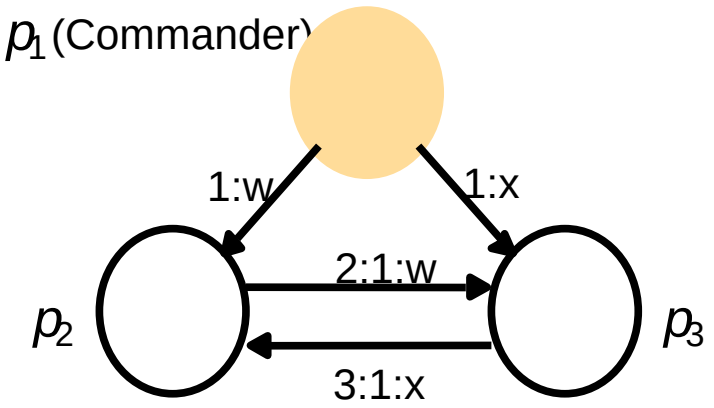
Impossibilidade com três processos

- Impossibilidade de solução para N=3 e f=1 [Lamport,Pease,Shostak82]. Casos:
  - Comandante (iniciador) é traidor
  - Um general é traidor

Obs: mensagem “3:2:a” significa “P3 diz que P2 diz que comando é a(ttack)” e a mensagem “3:2:w” significa “P3 diz que P2 diz que comando é w(ait)”



(A) General traidor



(B) Comandante traidor

© Pearson Education 2012

O Problema do Acordo Bizantino

- Esboço da Prova:
  - Em (A) C esta correto. Pelo Requisito de Integridade, P2 precisa decidir o valor recebido de C.
  - Como P2 não distingue as duas configurações, em (B) vai decidir também pelo valor recebido de C.
  - O mesmo se aplica ao P3 correto (que não sabe se C ou P2 é o traidor). Logo P3 sempre decidiria pelo valor recebido de C.
  - Mas então, no caso (B), P2 e P3 teriam decidido por valores distintos (w e a), o que contradiz o Requisito de Acordo.
  - Logo, não pode existir solução.
- Note que o problema todo está no fato de que no caso (A), o processo P3 forjou a informação recebida de C, e que P2 é incapaz de distinguir os dois casos.
- Obs: Se os generais usarem assinaturas digitais em suas mensagens (ou seja, as mensagens originais não puderem ser forjadas), então o PGB para N=3*f tem solução.

Impossibilidade para N <= 3*f

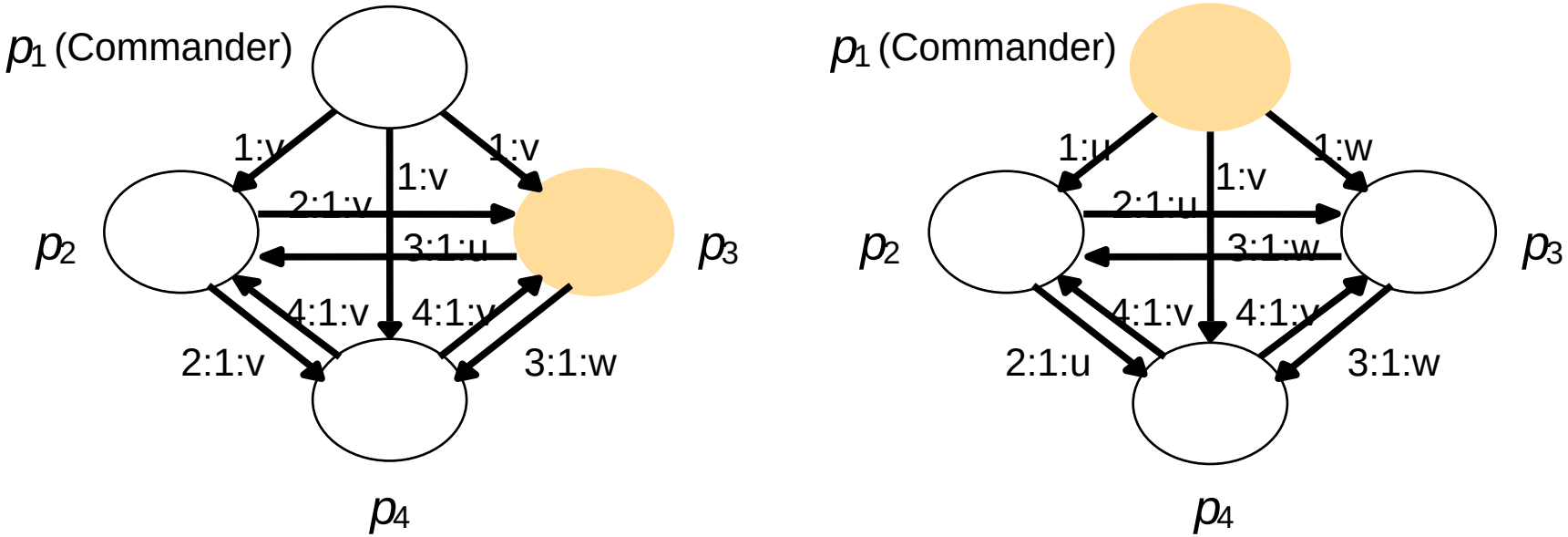
- Pease, Shostak e Lamport (1980) mostraram que o problema de impossibilidade do PGB para qualquer N ≤ 3*f é redutível ao problema N=3 e f=1. Ideia:
- Faça 3 processos P1,P2 e P3 ***simularem*** o comportamento de n1, n2, n3 generais, respectivamente, tal que:
  - n1+n2+n3 = N
  - n1, n2, n3 ≤ N/3 + 1
  - processos corretos simulam o comportamento de generais corretos
- Um dos processos só simula generais traidores
  - possível, pois como no máximo *f* generais são traidores e N ≤ 3*f e n1, n2, n3 ≤ N/3 +1, então:
  - No máximo *f* generais simulados são falhos.

O Problema dos Generais Bizantinos

- Como supostamente o programa de simulação dos generais está correto, a simulação termina.
- Suponha que os N – f generais corretos conseguissem chegar ao consenso.
- Neste caso, os processos simuladores correspondentes teriam chegado ao consenso (pegando o valor decidido pelos seus generais simulados) o que contradiz o resultado para N=3 com f=1.

Solução com um único processo falho

- Quatro generais bizantinos
  - Problema com N >= 3*f* +1, verificar em Pease *et. al.* 1980.
  - N=4, *f*=1



Faulty processes are shown coloured

© Pearson Education 2012

Solução com um único processo falho

- Vejamos os 2 casos:
  - (A): todos os corretos terão recebido N-2 versões do valor proposto por C e terão recebido valores inconsistentes do processo traidor (P3).
  - (B): todos os corretos terão recebido valores inconsistentes do Comandante, concluindo que este é o traidor.
- Em ambos os casos, aplicando-se a função *maioria** sobre os valores originalmente propostos pelo Comandante, chega-se
  - ao mesmo valor ou
  - valor default quando não há valor majoritário, p.ex. maioria(u,a,w) = w
- Portanto, cada um dos generais corretos irá tomar a mesma decisão (atacar ou esperar)

Discussão da eficiência

- Quantas rodadas de mensagens são necessárias? (Esse é um fatos para o tempo que o algoritmos demora a terminar.)
- Quantas mensagens são enviadas e de que tamanho? (Isso mede a utilização de largura de banda e tem um impacto sobre o tempo de execução.)
- O número de rodadas mínimo é f+1 [Fischer&Lynch82].
- No caso geral, (*f* >= 1), no algoritmo acima o número de mensagens é O(N^f+1), mas existem otimizações, em especial, se as mensagens contém assinatura digital [Dolev e Strong (1983)], pode-se obter número de mensagens = O(N^2) ;
- O alto custo do algoritmo só se justifica em sistemas em que há problemas de segurança. Se defeitos são de hardware, então falhas não são realmente arbitrárias.

Impossibilidade em sistemas assíncronos

- Se o sistema é assíncrono pode ser impossível distinguir um *crash* de uma mensagem que demora um tempo indeterminado.
  - Fischer, Lynch & Paterson (1985) apresentaram um resultado teórico fundamental → é impossível garantir a obtenção de consenso em um sistema assíncrono com falhas tipo *crash*
- Principais Razões:
  - Consenso com falha (*crash*, arbitrária) requer que cada processo receba um conjunto completo de valores de todos os demais processos corretos → só assim consegue criar uma base idêntica de valores para a tomada de decisão comum
  - Devido a inexistência de um limite superior para o tempo de comunicação, não é possível distinguir uma falha de processo (*fail-stop*), de uma mensagem muito demorada (falha de temporização)

Abordagens para lidar com a Impossibilidade

- Levando em conta que:
  - o consenso é um serviço fundamental e
  - a maioria dos sistemas são assíncronos
- Como resolver? Possíveis abordagens:
  - Eliminar a possibilidade de falhas "*fail-stop*" (mascarando falhas)
    - Ideia: Manter o estado atualizado de cada processo em memória persistente. Após uma falha, processo é reiniciado a partir do estado salvo. Os demais processos só percebem um atraso na execução do processo
  - Usar detectores de falha
    - Ideia: processos podem concordar em jugar falho u processo que não respondeu por mais do que um tempo limitado, Chandra e Toueg (1996).
  - Usar algoritmos randômicos (aleatoriedade)
    - Ideia: Comportamento randômico do algoritmo impede que o adversário esperto consiga sistematicamente impedir o consenso, Canetti e Rabin (1993).
    - Obs.: Não vai garantir que consenso seja sempre alcançado, mas com certa probabilidade (dependendo do sistema) e após tempo T, todos os processos saberão se consenso foi ou não atingido.