Computação Distribuída I

Prof. Lau Cheuk Lung, PhD
E-mail: lau.lung@inf.ufsc.br
Programa de Pós-Graduação em Ciência da Computação - PPGCC
Departamento de Informática e Estatística – INE
Universidade Federal de Santa Catarina - UFSC

Tópicos da aula

1. Algoritmo de eleição e exclusão mútua
2. Detecção de *deadlocks* (impasse)
3. Algoritmos de acordo (consenso distribuído)
4. Transações distribuídas
5. Memória compartilhada e distribuída

Algoritmos de eleição.

- Muitos algoritmos distribuídos requerem um processo para atuar como coordenador (ex: coordenador do algoritmo exclusão mutua centralizada; técnicas de replicação primário/backup e líder/seguidor), iniciador (ex: consenso), seqüenciador (difusão atômica) ou para desempenhar uma tarefa especial (ex: executar um processamento).
- Assumiremos que cada processo tem um número único, por exemplo, de P0 a PN onde o processo com o identificador de menor número é o mais antigo do grupo.
- Em geral, algoritmos de eleição tenta localizar o processo com o menor identificador (*rank*) para ser indicado como líder (ou coordenador). No entanto, os algoritmos de eleição se diferem no modo como fazem essa indicação.

Algoritmos de eleição.

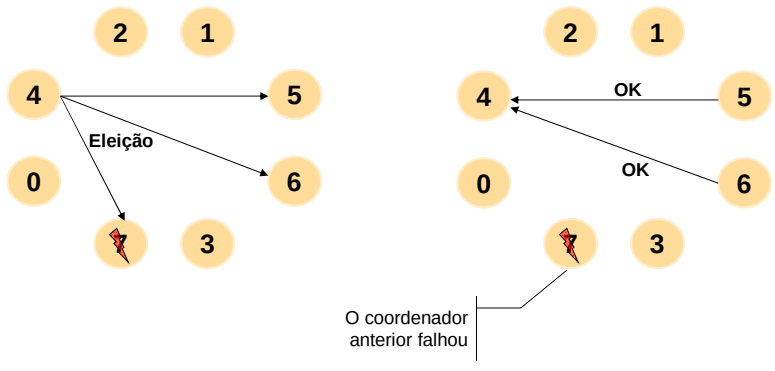
- Algoritmo 1: Bully (Garcia-Molina 1982)
- Quando um processo qualquer nota que o coordenador não mais responde a requisições, ele inicia uma eleição. Um processo Pi prepara uma eleição da seguinte forma:
  - Procedimento:
    - Pi envia uma mensagem "ELEIÇÃO" para todos os processos com número superior ao seu;
    - Se nenhum deles responde, Pi vence a eleição e torna-se o novo coordenador.
    - Se um ou mais responde, o novo coordenador deverá ser aquele com o maior identificador. A tarefa de Pi termina.

Algoritmos de eleição.

- Em qualquer momento, um processo Pj (sendo j > i) pode receber uma mensagem ELEIÇÃO de um de seus colegas com número identificador inferior ao seu. Quando isso ocorre, tal processo Pj simplesmente envia uma mensagem "OK" ao emissor da mensagem "ELEIÇÃO" indicando que ele está vivo e pode assumir o comando.
- Então, se existir outro(s) processo(s) Pk, tal que (k > j), o processo Pj envia uma nova mensagem "ELEIÇÃO" a esse(s) processo(s) Pk, repetindo o procedimento acima. Todos os processos desistirão menos um (o de maior identificador), esse será o coordenador.
- Ele anuncia sua vitória enviando uma mensagem "COORDENADOR" para todos processos que ele está iniciando as atividades de coordenador.

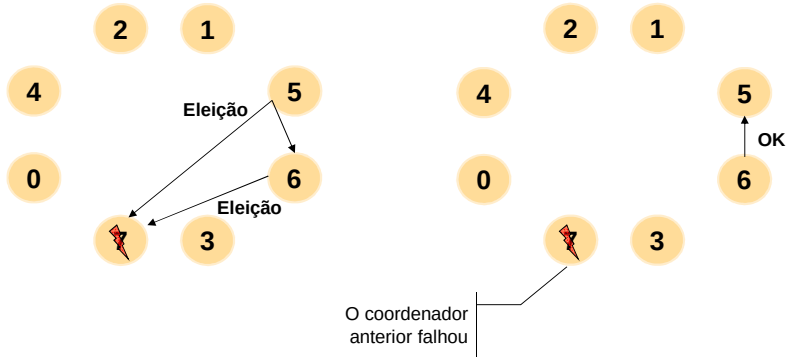
Algoritmos de eleição.

- Passos do algoritmo de eleição bully



Algoritmos de eleição.

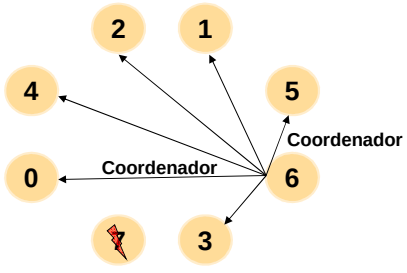
- Passos do algoritmo de eleição bully



4

Algoritmos de eleição.

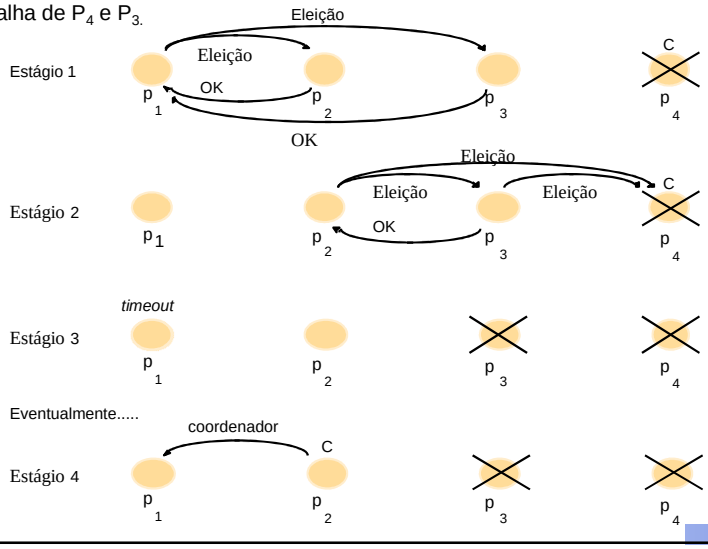
- Passos do algoritmo de eleição bully



Algoritmos de eleição.

- Algoritmo de eleição Bully

A eleição do coordenador P_2,
Após a falha de P_4 e P_3.

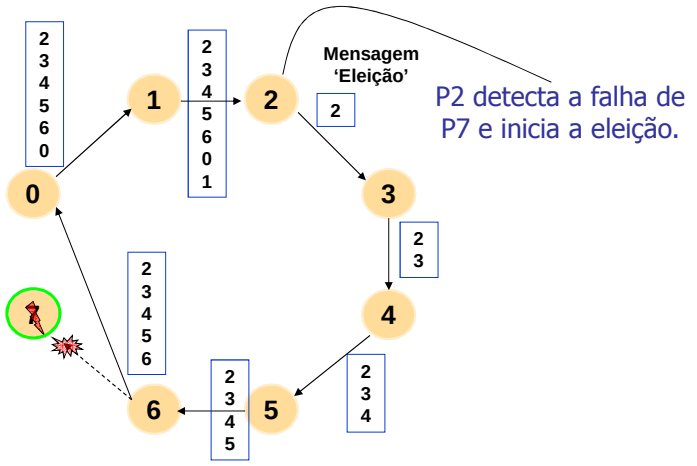


Algoritmos de eleição.

- Algoritmo 2: Anel (Chang-Roberts 1979)
  - Esse algoritmo é baseado no uso de um anel virtual, mas sem *token*. Assumimos que os processos são fisicamente ou logicamente ordenado de tal forma que cada processo sabe quem é seu sucessor.
  - Quando qualquer processo nota que o coordenador não está funcionando, ele constrói uma mensagem ELEIÇÃO contendo seu próprio número de processo (identificador) e envia a mensagem para o seu sucessor. Se o sucessor está fora do ar, o emissor pula para o próximo membro do anel virtual e assim sucessivamente até encontrar um ativo.
  - Em cada passo, quando um processo ativo recebe a mensagem ELEIÇÃO ele inclui também seu identificador na mensagem e envia esta para o seu sucessor.

Algoritmos de eleição.

- Algoritmo 2: Anel (Chang-Roberts 1979)



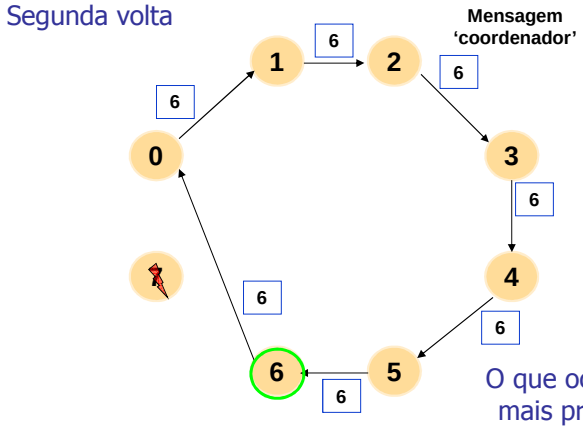
6

Algoritmos de eleição.

- Algoritmo 2: Anel (Chang-Roberts 1979)
  - Em algum momento, a mensagem ELEIÇÃO retornará ao processo que iniciou tudo.
    - Esse processo identifica este evento pelo conteúdo da mensagem, encontrando lá o seu próprio identificador.
  - A partir daí, a mensagem é modificada para COORDENADOR e volta a circular entre os processos do anel virtual indicando quem é o novo coordenador (usualmente o processo de maior número).

Algoritmos de eleição.

- Algoritmo 2: Anel (Chang-Roberts 1979)



O que ocorre quando dois ou mais processos detectam a falha do coordenador ?

Exclusão mútua

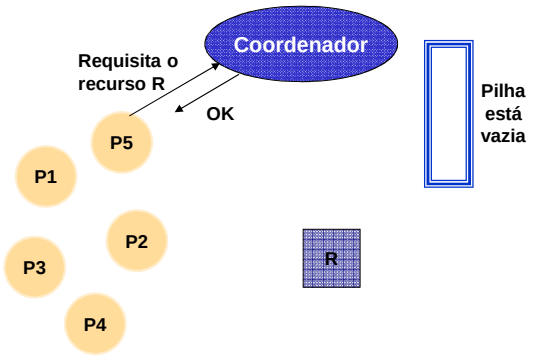
- Num sistema distribuídos existem recursos que não pode ser acessados simultaneamente por diferentes processos se desejarmos o correto funcionamento de um programa. Para evitar isso, é necessário mecanismos para garantir o acesso exclusivo dos recursos – chamamos isso de exclusão mútua.
- Propriedade:
  - Exclusão mútua: dado um recurso que pode ser acessado por diferentes processos ao mesmo tempo, somente um processo por vez pode acessar esse recurso. Contudo, um processo que ganha acesso a um recurso deve liberá-lo para que outro processo ganhe acesso ao mesmo.
  - Starvation: qualquer processo que requisita um recurso deve recebê-lo em qualquer momento.

Exclusão mútua

- Algoritmo centralizado:
  - Um processo do sistema é indicado como coordenador (ver algoritmo de eleição de líder) para controlar o acesso a um região crítica (RC). Qualquer processo que deseja entrar em uma RC deve pedir ao coordenador. Se o coordenador verificar que nenhum processo está acessando a RC então, o processo solicitante obtém o acesso. Caso contrário, este processo deve esperar até que o processo que está acessando a RC libere o recurso e avise o coordenador.

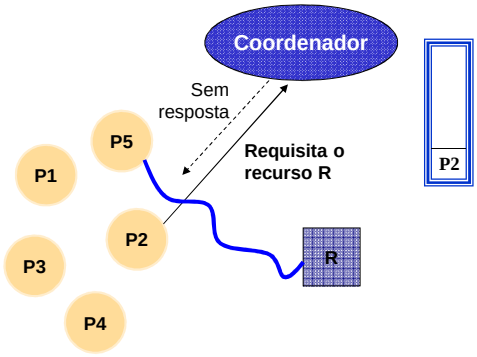
Exclusão mútua

- Algoritmo centralizado – exemplo:



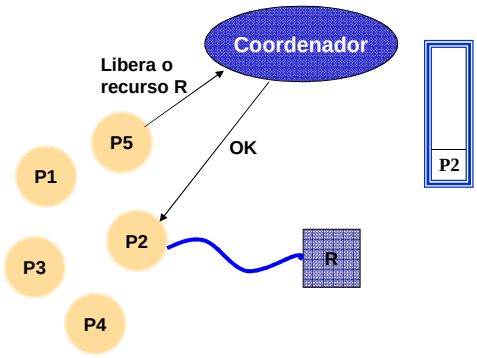
Exclusão mútua

- Algoritmo centralizado – exemplo:



Exclusão mútua

- Algoritmo centralizado – exemplo:



Exclusão mútua

- Algoritmo distribuído:
  - Quando um processo deseja entrar na RC ele deve enviar uma mensagem para o sistema (todos os outros processos que podem acessar a RC). A mensagem M tem o seguinte conteúdo:
    - Id: identificador do processo;
    - Rc: Nome da RC que deseja acessar;
    - Ts: uma estampinha de tempo único (timestamp) gerado pelo processo – baseado no algoritmo de Lamport78.

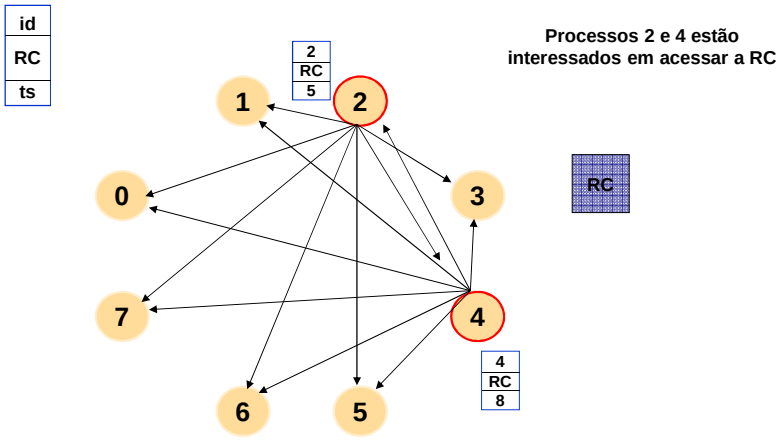
Mensagem M

Exclusão mútua

- Algoritmo distribuído:
  - Qualquer processo ao receber esta mensagem deve responder ao emissor dizendo OK (garantindo o acesso ao recurso) ou não envia nada se:
    - Ele é o processo que está usando a RC, então ele enfileira a mensagem M e não responde.
    - Ele não está usando a RC mas, está esperando sua vez. Ele compara o timestamp da mensagem M com o seu. Se for menor que o seu então envia uma resposta OK, caso contrário, ele enfileira M e não responde. -> rever Lamport78
  - Todo processo ao terminar de usar a RC deve enviar um OK aos processos solicitantes.
  - Um processo solicitante pode acessar a RC quando tiver o OK de todos os processos do sistema.

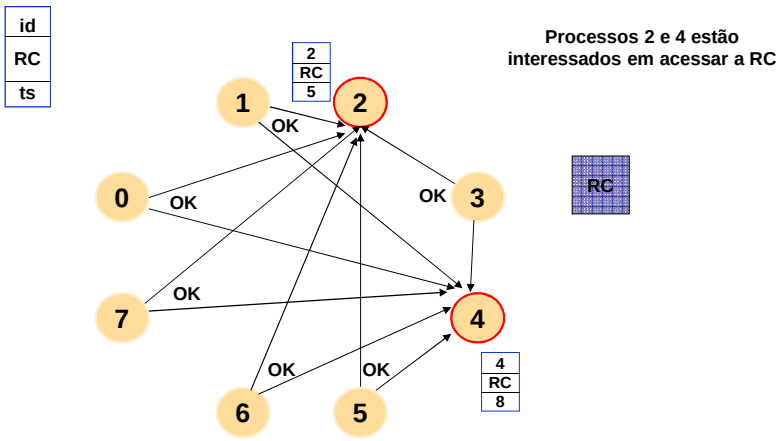
Exclusão mútua

- Algoritmo distribuído - exemplo:



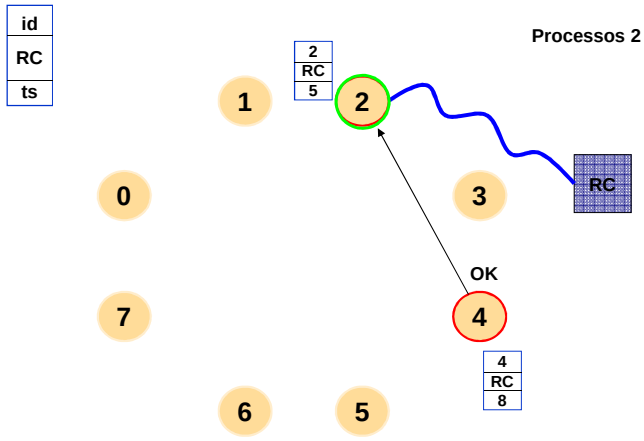
Exclusão mútua

- Algoritmo distribuído - exemplo:



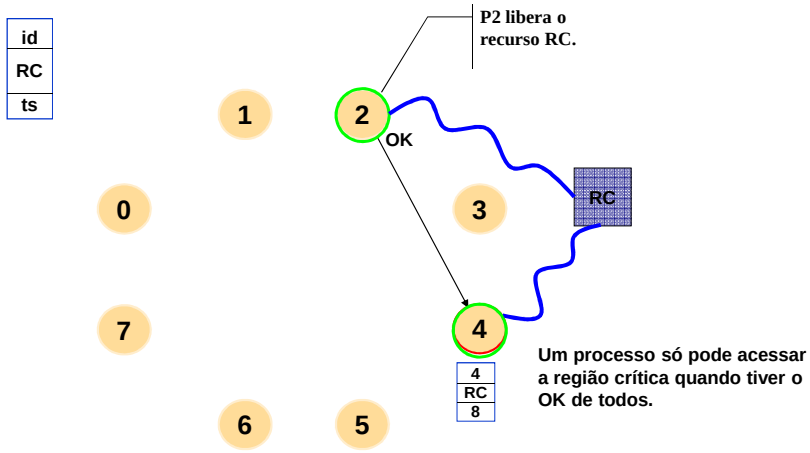
Exclusão mútua

- Algoritmo distribuído - exemplo:



Exclusão mútua

- Algoritmo distribuído - exemplo:

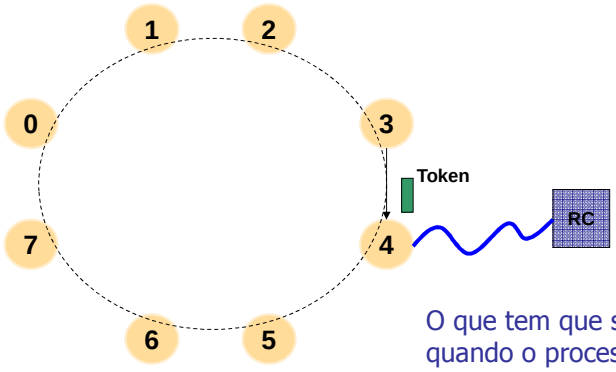


Exclusão mútua

- Algoritmo baseado no token
  - Neste algoritmo é definido um anel virtual em que um token circula entre os processos do sistema. Um processo de posse do token tem direito de acesso a uma determinada RC.
  - Se ele não deseja acessar a RC o token deve ser passado adiante.

Exclusão mútua

- Algoritmo baseado no token - exemplo



O que tem que ser feito quando o processo que está com token falha?

Detecção de deadlock em sistemas distribuídos

- Num sistema distribuídos existem recursos que não pode ser acessados simultaneamente por diferentes processos se desejarmos o correto funcionamento de um programa. Para evitar isso, é necessário mecanismos para garantir o acesso exclusivo dos recursos – chamamos isso de exclusão mútua.
- Características do deadlock, condições necessárias:
  - **Exclusão mútua**: existência de pelo menos um recurso não compartilhável no sistema, isto é, somente um processo por vez pode utilizar esse recurso;
  - **Segura espera**: existência de um processo que está utilizando um recurso e está a espera por outros recursos que estão sendo utilizados por outros processos;

Detecção de deadlock em sistemas distribuídos

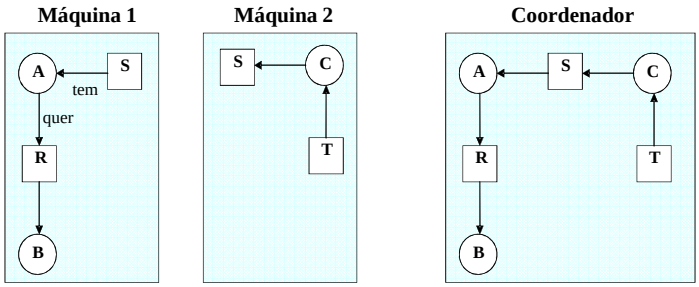
- **Recurso não preemptados**: um recurso só pode ser liberado (desalocado) de forma voluntária pelo processo que está utilizando esse recurso;
- **Espera circular**: existência de um conjunto de processos (P0, P1, P2, ... Pn) bloqueados, tal que P0 espera por um recurso mantido por P1**,** P1 espera por um recurso mantido por P2, ..., Pn-1 espera por um recurso mantido por Pn e, por fim, Pn espera por um recurso mantido por P0.

Detecção de deadlock centralizado

- Cada máquina mantém um grafo de alocação local de seus recursos pelos processos distribuídos.
- Um coordenador central recebe esses grafos locais (através de troca e mensagens) e assim constrói um grafo global (a união dos grafos locais).
- Os grafos locais são enviados pelas máquinas locais ao coordenador toda vez que um recurso é alocado ou desalocado no sistema, ou seja, sempre que um arco é incluído ou excluído do grafo local.

Detecção de deadlock centralizado

- Detecção de deadlock baseado em grafo de alocação de recursos.



Mensagens enviadas pela máquina 1:

A tem S

A quer R

B tem R

Mensagens enviadas pela máquina 2:

C tem T

C quer S

Processo

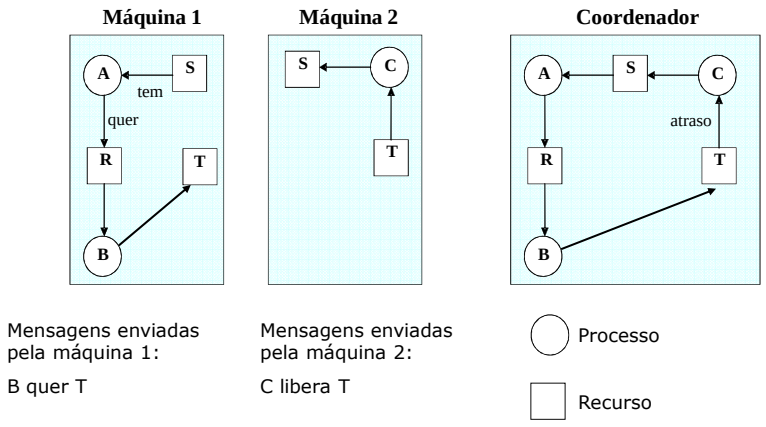
Recurso

Detecção de deadlock centralizado

- Quando o coordenador verifica um ciclo (espera circular) um *deadlock* é detectado e, para que o *deadlock* seja removido, o coordenador decide por matar um dos processos causadores do *deadlock* – um processo de maior ou menor índice.
- Como essas informações são enviadas através da rede pode ocorrer atrasos causando inversão na ordem de entrega das mensagens, causando dessa forma um falso *deadlock*.

Detecção de deadlock centralizado

- Falso deadlock



Detecção de deadlock distribuído

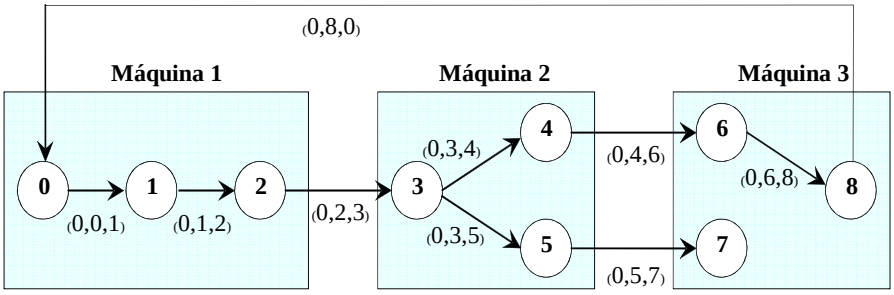
- Nesse algoritmo não existe o papel de um coordenador central. Quando um processo Pi suspeita de um deadlock (ex: espera por um recurso que não é desalocado por um outro processo Pj após um longo tempo, ex: *timeout*) ele monta uma mensagem *probe* contendo:
  - O número do processo que está bloqueado;
  - O número do processo que está enviando a mensagem (origem);
  - O número do processo para quem a mensagem é destinada (destino).

Detecção de deadlock distribuído

- Quando esta mensagem chega ao processo destino Pj, este verifica se ele também está esperando por algum outro recurso que está em uso por um outro processo.
  - Se sim, então Pj modifica o segundo e terceiro campo da mensagem, sempre mantendo o primeiro, com o seu identificador e o identificador do processo destino que ele está esperando desbloquear o recurso, respectivamente.
  - Da mesma forma, se o processo Pj estiver esperando por vários recursos mantido por diferentes processos então a mensagem *probe* é enviada para cada um desses processos.

Detecção de deadlock distribuído

- Caso a mensagem *probe* dê uma volta e retorne ao processo originador (nesse caso, o processo Pi) da mensagem – identificado pelo primeiro campo da mensagem *probe* – a sistema está em deadlock e algum processo deve se matar (ex: o maior ou menor identificador).



Transações distribuídas

- Transação atômica
  - Todas as técnicas de sincronização vistas até agora são essencialmente de baixo nível, tal como semáforos.
    - Eles requerem que o programador esteja estreitamente envolvido com detalhes de exclusão mútua, gerenciamento de região critica, prevenção de deadlock, e recuperação de falhas.
    - O que realmente queremos é uma abstração de alto nível, que esconda estes aspectos técnicos e permita ao programador concentrar nos algoritmos e como os processos trabalham junto em paralelo.

Transações distribuídas

- Transação atômica
  - Em comunicações cliente-servidor as requisições são raramente constituídas de operações únicas, usualmente, são varias requisições para realizar uma única tarefa.
  - Além disso, o servidores são capazes de atender vários clientes simultaneamente. Os algoritmos de transação atômica visam garantir a integridade dos dados e consistência.
    - Para isso, o conjunto de requisições para realização de uma única tarefa são encapsulada em uma transação.
  - Do ponto de vista do programador, a transação é executada como um programa seqüencial, mesmo que possa executar de forma concorrente com outros programas ou que falhas ocorram durante sua execução.

Transações distribuídas

- Exemplo de uso
  - Aplicação bancaria que atualiza uma base de dado num servidor remoto. O cliente chama um servidor bancário usando um PC com um modem com a intenção de retirar dinheiro de uma conta A e depositá-lo em um conta B. A operação é executada em dois passos:
    - retirar(quantidade, conta A) ;
    - depositar(quantidade, conta B);
  - Se, por algum motivo, a conexão telefônica cai após a primeira requisição mas antes da segunda, a primeira conta terá sido debitada mas a segunda não terá recebido o crédito. O dinheiro simplesmente desaparece.

Transações distribuídas

- Exemplo de uso
  - Ser capaz de agrupar estas duas operações em uma transação atômica resolveria o problema. Ou ambas requisições seriam completadas, ou nenhuma seria completada. A chave para isso é o retorno ao estado inicial (*rolling back*) da transação se esta falha antes de completar.
  - Outro exemplo, reserva de uma passagem aérea:
    - reserva Curitiba -> São Paulo (TAM)
    - reserva São Paulo -> Madrid (Varig)
    - reserva Madrid -> Frankfurt (Spanair)
    - reserva Frankfurt -> Moscou (Lufthansa) .

Transações distribuídas

- Definição de transação:
  - É uma **seqüência de operações** que devem ser tratadas como uma unidade atômica (indivisível), executadas em seqüência e sem interrupções ou que possam ser executadas de forma concorrente, mas com os mesmos resultados de uma execução seqüencial.
- Primitivas de um serviço de transação:
  - BEGIN_TRANSACTION: marca o inicio de uma transação;
  - END_TRANSACTION: termina a transação e tenta confirmar (*commit*) ;
  - ABORT_TRANSACTION: mata a transação; restaura o valor antigo;
  - READ: lê dados de um arquivo (ou outro objeto);
  - WRITE: escreve dados em um arquivo (ou outro objeto).

Transações distribuídas

- Propriedades da transação atômica (ACID)
  - **A**tomicidade (*Atomicity*): garante a propriedade tudo ou nada, ou todas as operações são confirmadas ou abortadas. Se uma transação aconteceu, ele ocorreu em uma ação instantânea e indivisível. A garantia dessa propriedade é alcançada com os protocolos de confirmação (*commitment*);
  - **C**onsistência (*Consistency*): garante que o resultado de uma transação é correta, não viola as propriedades invariantes do sistema. A consistência dos dados no cliente e no servidor deve ser garantida quando a transação é confirmada ou mesmo quando é abortada. Durante a execução da transação é permitida que as propriedades invariantes do sistema possam ser violada, por isso os estados intermediários não podem ser visíveis aos outros processos. Exemplo anterior, lei da conservação do dinheiro;

Transações distribuídas

- Propriedades da transação atômica (ACID)
  - Durabilidade (*Durability*): refere ao fato que uma vez a transação é confirmada, não importa o que aconteça, a transação vai adiante e os resultados tornam-se permanentes. Após a confirmação da transação (*commit*), nenhuma falha pode desfazer os resultados da transação ou causar sua perda. Essa propriedade pode ser alcançada com protocolo de recuperação. Os dados e transações pendentes podem ser armazenados num meio de armazenamento persistente.
  - Isolamento (*Isolation*): se duas ou mais transações estão sendo executadas ao mesmo tempo, para cada um delas e aos outros processos, o resultado final é como se todas as transações tivessem sido executadas seqüencialmente. Independentemente se as transações são concorrentes ou não, cada transação é executada sem interferência de outras transações. A propriedade de isolamento pode ser assegurada por mecanismos de controle de concorrência;

Transações distribuídas

- Propriedades da transação atômica (ACID)
  - Durabilidade (*Durability*): refere ao fato que uma vez a transação é confirmada, não importa o que aconteça, a transação vai adiante e os resultados tornam-se permanentes. Após a confirmação da transação (*commit*), nenhuma falha pode desfazer os resultados da transação ou causar sua perda. Essa propriedade pode ser alcançada com protocolo de recuperação. Os dados e transações pendentes podem ser armazenados num meio de armazenamento persistente.
  - Isolamento (*Isolation*): se duas ou mais transações estão sendo executadas ao mesmo tempo, para cada um deles e aos outros processos, o resultado final é como se todas as transações tivessem sido executadas seqüencialmente. Independentemente se as transações são concorrentes ou não, cada transação é executada sem interferência de outras transações. A propriedade de isolamento pode ser assegurada por mecanismos de controle de concorrência;

Transações distribuídas

- Exemplo:

```
BEGIN_TRANS   BEGIN_TRANS    BEGIN_TRANS
 X = 0;        X = 0;         X = 0;
 X = X + 1;    X = X + 2;     X = X + 3;
 END_TRANS     END_TRANS      END_TRANS
```

- Dependendo de qual transação termina por último X pode ser 1, 2 ou 3.
- Nunca X = 4, 5, 6 ou qualquer outro valor que nao seja 1, 2 ou 3.

Transações distribuídas

- Exemplo2: SaldoA = $100, SaldoB = $200 e SaldoC = $300

Correto

Transação T:		Transação U:	
saldo = b.getSaldo();		*saldo = b.getSaldo();*	
*b.deposito(saldo*1.1);*		*b.deposito(saldo*1.1);*	
a.retirada(saldo/10)		*c.retirada(saldo/10)*	
saldo = b.getSaldo();	$200		
*b.deposito(saldo*1.1);*	$220		
a.retirada(saldo/10)	$80		
		saldo = b.getSaldo();	$220
		*b.deposito(saldo*1.1);*	$242
		c.retirada(saldo/10)	$278

Transações distribuídas

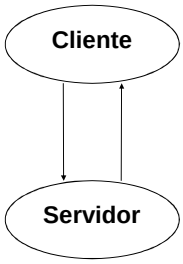
- Execução incorreta

Incorreto: execução concorrente

saldo = b.getSaldo();		*saldo = b.getSaldo();*	
*b.deposito(saldo*1.1);*		*b.deposito(saldo*1.1);*	
a.retirada(saldo/10)		*a.retirada(saldo/10)*	
saldo = b.getSaldo();	$200		
		saldo = b.getSaldo();	$200
		*b.deposito(saldo*1.1);*	$220
*b.deposito(saldo*1.1);*	$220		
a.retirada(saldo/10)	$80		
		c.retirada(saldo/10)	$280

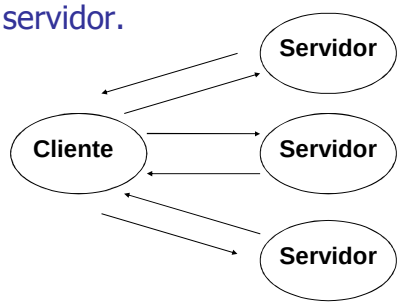
Transações distribuídas

- Os tipos de transação existentes são:
  - transação simples: modelo em que um cliente invoca um servidor para executar um conjunto de operações de uma transação;



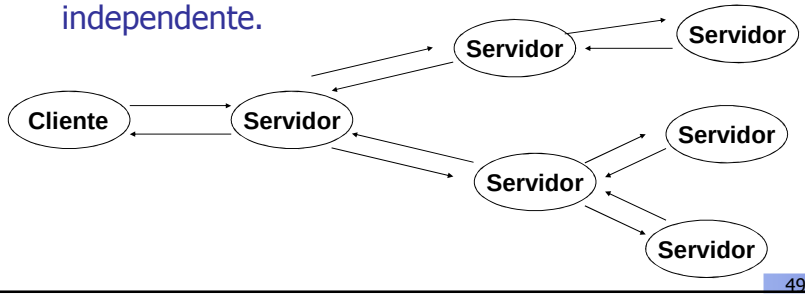
Transações distribuídas

- Os tipos de transação existentes são:
  - transação distribuída simples: modelo em que um cliente invoca vários servidores necessários para completar uma transação requisitada. Nesse caso, cada operação é executada em cada servidor serialmente. Se for várias operações em cada servidor, então as operação referentes a um devem ser executadas antes de ir para o próximo servidor.



Transações distribuídas

- Os tipos de transação existentes são:
  - transação aninhada: modelo em que um cliente invoca um ou mais servidores que por sua vez invoca(m) outros servidores e assim sucessivamente. Um servidor invocando outro servidor pode usar a primitiva *fork*. Nesse modelo, transações podem ser executadas concorrentemente em diferentes servidores, podendo também ser confirmadas ou abortadas de forma independente.



Transações distribuídas

- Writeaheadlog
  - As vezes chamado de lista de intenções, é um método para implementar transações. É baseado em logs, somente após o log ter sido escrito com sucesso, a mudança é feita no arquivo. A figura abaixo dá um exemplo de como o log trabalha. Temos uma simples transação que usa duas variáveis compartilhada (ou objetos), *X* e *Y*, ambos iniciados em 0. Para cada uma das três instruções dentro da transação, um registro de log é escrito antes de executar o instrução, dando o velho e o novo valor, separado por barra.

Transações distribuídas

- Writeaheadlog

```
X = 0;                Log        Log        Log
Y = 0;                X = 0/1    X = 0/1    X = 0/1
BEGIN_TRANS                      Y = 0/2    Y = 0/2
  X = X + 1;                                X = 1/2
  Y = Y + 2:
  X = Y * X;
END_TRANS
```

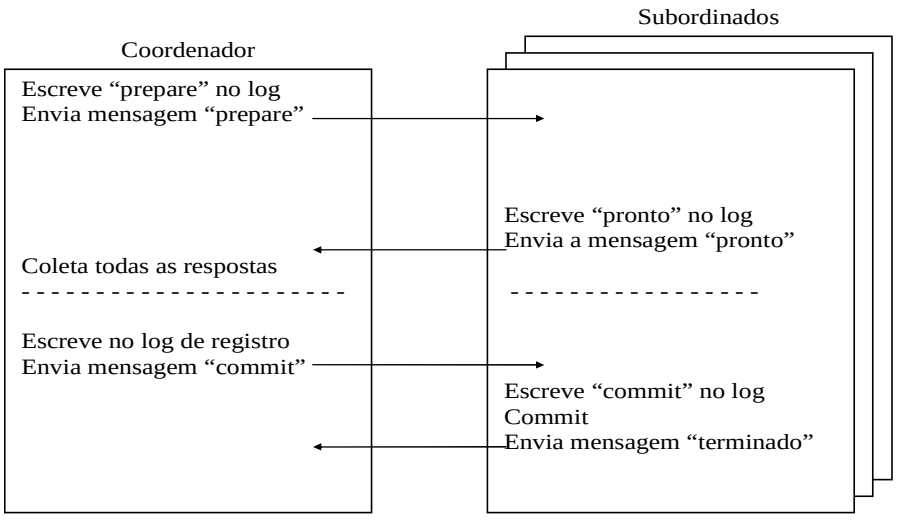
  - Se a transação tem sucesso e é confirmada, um registro de *commit* é escrito para o log. Se a transação é abortada, é possível obter o estado das variáveis antes da transação a partir dos logs. Esse processo é chamado *rollback*

Transações distribuídas

- Protocolo de confirmação (*commit*) de duas fases (proposto por Gray, 1978)
  - Esse protocolo não é o único, mas é o mais utilizado. Um dos processos envolvido na transação faz a função de coordenador, usualmente, é ele quem cuida da transação. O protocolo inicia quando o coordenador escreve um log de entradas dizendo que ele está iniciando o protocolo de confirmação, seguido pelo envio de uma mensagem aos outros processos participantes da transação dizendo para se prepararem pro *commit*.

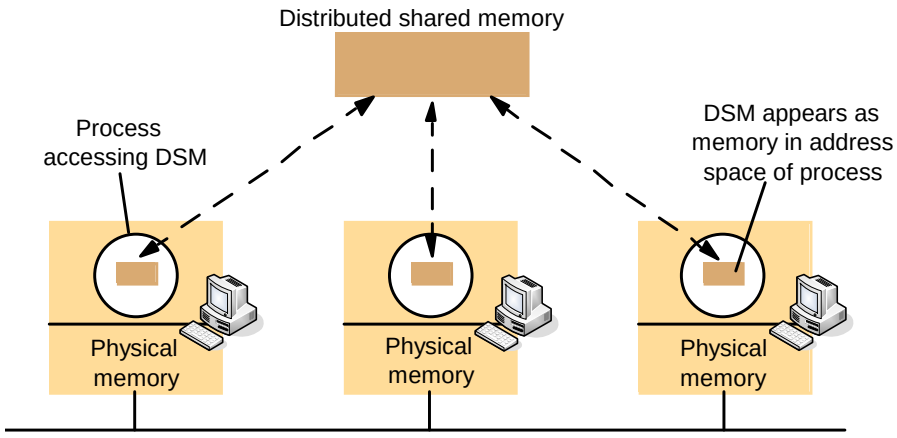
Transações distribuídas

- Protocolo de confirmação (*commit*) de duas fases



Memória compartilhada

- Abstração de memória compartilhada distribuída (figura 16.1)



Memória compartilhada

- Características Gerais
  - Computadores não compartilham memória física
  - Abstração mais simples que passagem de mensagens
  - Cada processo faz acesso à DSM como se a memória pertencesse ao seu espaço de endereçamento
  - Um subsistema deve garantir que as modificações feitas por um processo são observadas pelos demais
  - Dificuldade de implementação: bom desempenho e escalabilidade
  - Volume de comunicação na rede depende do protocolo de consistência da DSM
  - Ideal para programação paralela e de grupos

DSM versus Passagem de mensagem

- Marshalling: processos usando DSM compartilham variáveis diretamente, sem necessidade de marshalling
  - processos podem interferir um no outro, causando falha
  - como resolver o problema de diferentes representações?
- Sincronização: através de construtores de linguagem de programação (locks e semáforos)
- Persistência: DSM pode sobreviver além do término dos processos
- Eficiência: bastante variável; depende do padrão de acesso a variáveis; normalmente passagem de mensagens é mais eficiente

Abordagens de Implementação de um DSM

- Hardware-based: Clusters de processadores e módulos de memória conectados por uma rede de alta velocidade. Exemplo: Dash
- Page-based: Uma região de memória virtual que ocupa o mesmo intervalo de endereçamento em cada processo participante; cada núcleo mantém a consitência da DSM. Exemplos: Ivy, Munin, Clouds, Choices
- Middleware: Linguagens de programação (ou extensões) fornecem abstrações que são implementadas através de chamadas a rotinas de biblioteca inseridas por um compilador, causando comunicação entre os processos. Exemplo: Orca, Linda, JavaSpace, TSpaces

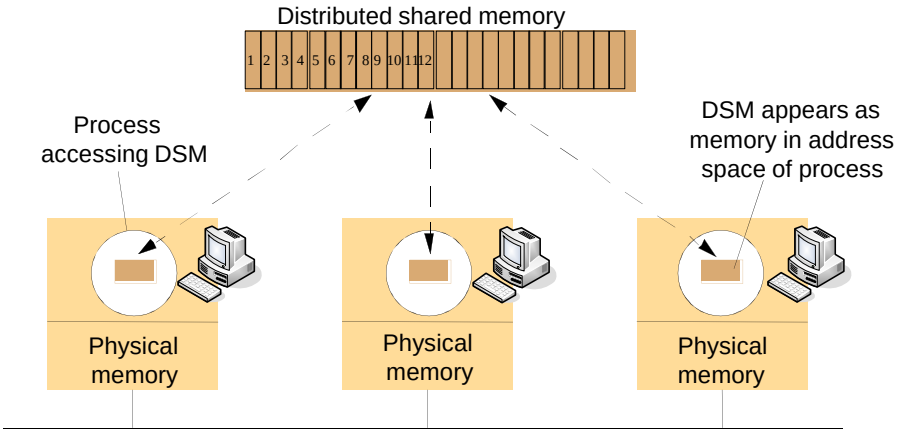
Abordagens de Implementação de um DSM

- Hardware-based: Clusters de processadores e módulos de memória conectados por uma rede de alta velocidade. Exemplo: Dash



Abordagens de Implementação de um DSM

- Page-based: Uma região de memória virtual que ocupa o mesmo intervalo de endereçamento em cada processo participante; cada núcleo mantém a consitência da DSM. Exemplos: Ivy, Munin, Clouds, Choices



Estruturas de DSM

- Byte-oriented: Permite aplicações (e implementações de linguagens) definir qualquer estrutura de dados sobre a memória compartilhada; o acesso a DSM é equivalente a acesso a memória virtual. Exemplo: Ivy e Mether
  - Duas operações apenas: *read* (ou *LOAD*) e *write* (ou *STORE*)
  - Se *x* e *y* são duas posições de memória, temos:
    - R(x)a: lê o valor *a* de uma posição *x*;
    - W(x)b: armazena o valor *b* na posição *x*;
  - Exemplo: W(x)1, R(x)2
    - Algum outro processo gravou 2 na posição x

Estruturas de DSM

- Shared objects: Memória compartilhada é vista como uma coleção de objetos compartilhados; sincronização é feita a nível de operação de objetos. Exemplo: Orca
  - Objetos em nível de linguagem;
  - Acessada através de invocação de métodos em objetos, nunca pelo acesso direto às suas variáveis membros;

Estruturas de DSM

- Immutable data: Coleção de dados imutáveis que todos os processos podem ler; modificação ocorre através de substituição do dado. Exemplo: Linda (modelo de tuplas), TSPace e JavaSpace;
  - Tuplas consistem em uma sequência de um ou mais campos de dados tipados, exemplo:
    - <"fred", 1958>, <"sid", 1964> e <4, 9.8, "yes">
  - Processos compartilham dados acessando esse espaço de tuplas usando as seguintes operações:
    - *write*: adiciona uma tupla sem afetar as tuplas existentes no espaço;
    - *read*: retorna o valor de uma tupla sem afetar o conteúdo do espaço de tuplas;
    - *take*: também retorna uma tupla, mas remove a tupla do espaço;

Estruturas de DSM

- *read* e *take* são bloqueantes, ou seja, ficam bloqueadas até retornar uma tupla que combine com a especificação.
- Nas operações *read* e *take* devem ser fornecidas uma especificação da tupla, e o espaço retorna QUALQUER tupla que combine com essa especificação, exemplo:
  - Tuplas no espaço: <"fred", 1958>, <"sid", 1964> e <4, 9.8, "yes">
  - *read*(<String, integer>) deve retornar <"fred", 1958> **ou** <"sid", 1964>
  - *take*(<String, 1958>) deve retornar <"fred", 1958>
- Implementando um contador em DSM
  - <s, count>:= myTS.take(<"counter", integer>);
  - myTS.write(<"counter", count+1>)

Memória compartilhada

- Programa no sistema Mether

```
#include "world.h"
struct  shared { int a,b; };
Program Writer:
  main() {
     struct shared *p;
     methersetup();        /* Initialize the Mether run-
  time */
     p = (struct shared *)METHERBASE;
                          /* overlay structure on METHER
  segment */
     p->a = p->b = 0;    /* initialize fields to zero */
     while(TRUE) {       /* continuously update structure
  fields */
        p ->a = p ->a + 1;
        p ->b = p ->b - 1;
     }
  }
//continua no próximo slide
```

Memória compartilhada

- Programa no sistema Mether

```
Program Reader:
  main() {
     struct shared *p;
     methersetup();
     p = (struct shared *)METHERBASE;
     while(TRUE) { /* read the fields once every second */
        printf("a = %d, b = %d\n", p –>a, p –>b);
        sleep(1);
     }
  }
```


DSM - Modelo de Consitência

- Consistência Sequêncial
  - todas as operações (leituras e escritas) de um certo processo são satisfeitas na mesma ordem do programa
  - as operações de memória pertencentes a processos distintos ocorrem em alguma ordem serial (isso não garante que a ordem absoluta seja respeitada)
- Consistência Fraca: explora conhecimento de sincronização entre operações para relaxar consistência. Exemplo: Quando um região crítica é bloqueada por um certo processo, não é necessário propagar modificações até que o bloqueio termine.

Memória compartilhada

- Dois processos acessando variáveis compartilhadas

Valores iniciais: a = b = 0

Process 1

```
br := b;
ar := a;
if(ar ≥ br) then
      print ("OK");
```

Process 2

```
a := a + 1;
b := b + 1;
```

p1: br := 0 e ar:=0 ←

p2: a := 1 ←

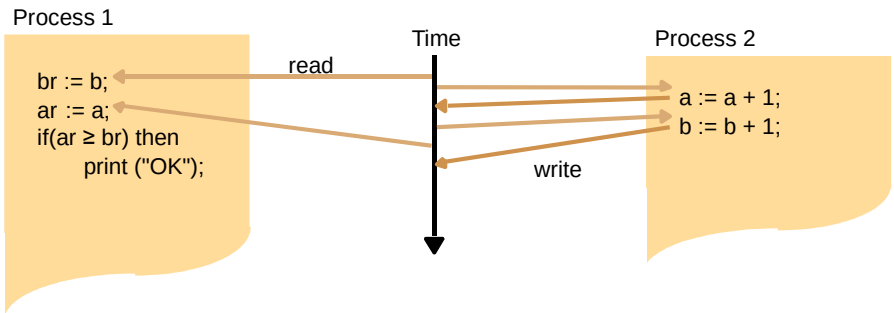
p1: br := 0 e ar:=1 ←

p2: b:=1 ←

p1: br := 1 e ar:=1 ←

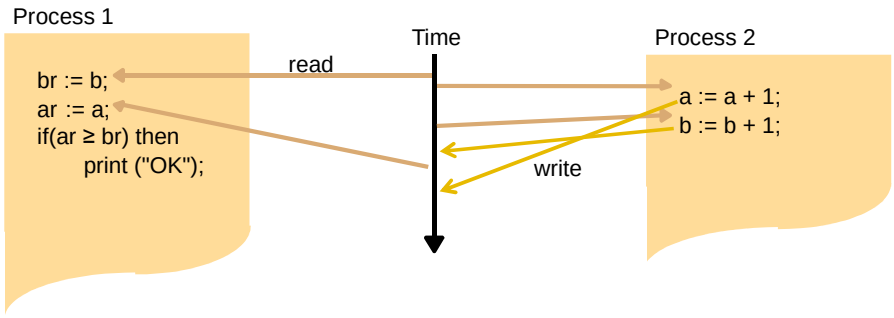
Memória compartilhada

- Intercalando na consistência seqüencial



Memória compartilhada

- Intercalando na consistência seqüencial



p1: br := 0 e ar:=0

p2: b := 1

p1: br := 1 e ar:=0

p2: a:=1

p1: br := 1 e ar:=1

Memória compartilhada

- Processos executando com consistência relaxada

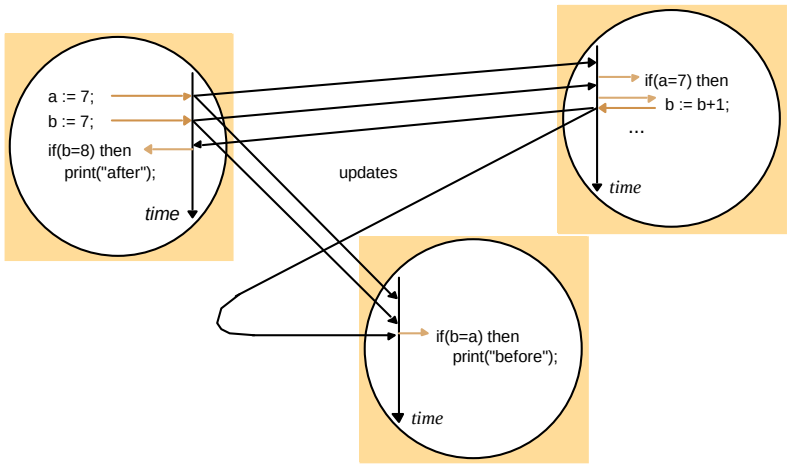
```
Process 1:
  acquireLock();              // enter critical section
  a := a + 1;
  b := b + 1;
  releaseLock();              // leave critical section
Process 2:
  acquireLock();              // enter critical section
  print ("The values of a and b are: ", a, b);
  releaseLock();              // leave critical section
```

Memória compartilhada

- Opções de Updates
  - Write-update: os updates realizados por um processo são feitos localmente e em seguida ocorre um multicast (ordenado) para todos os outros gerenciadores de réplica que contêm uma cópia do item de dado modificado; *multiple-reader-multiple-writer sharing*.
  - Write-invalidate: quando um processo quer modificar um item, primeiramente ocorre um multicast para todas as cópias para invalidá-las; updates são propagados somente quando ocorre uma leitura do item; *multiple-reader-single-writer sharing.* (Thrashing: quando o sistema gasta mais tempo invalidando e transferindo itens que trabalhando).

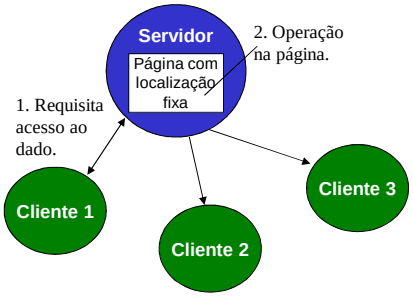
Memória compartilhada

- Figura 16.5: DSM usando write-update

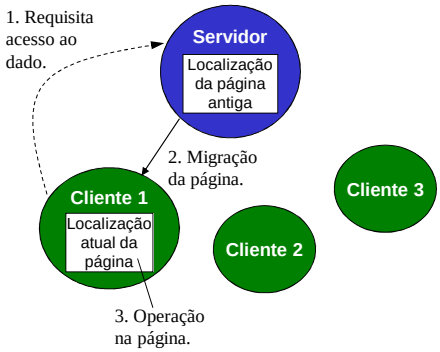


Arquitetura de DSM

- Centralizada

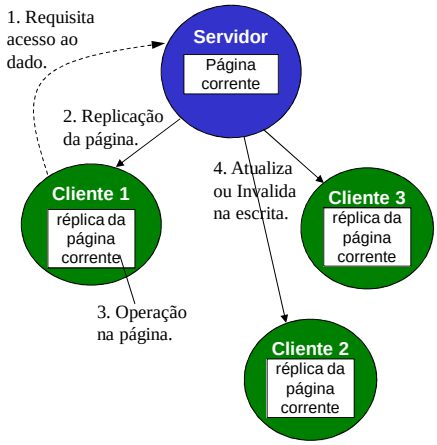


- Migração

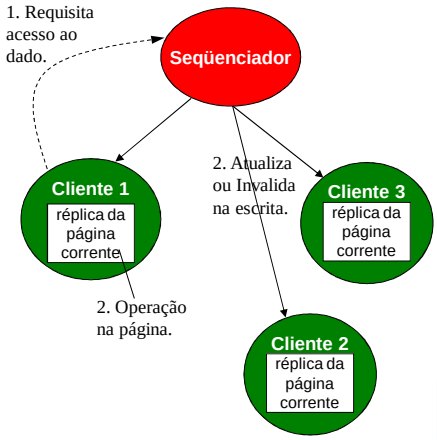


Arquitetura de DSM

- Replicação só-leitura

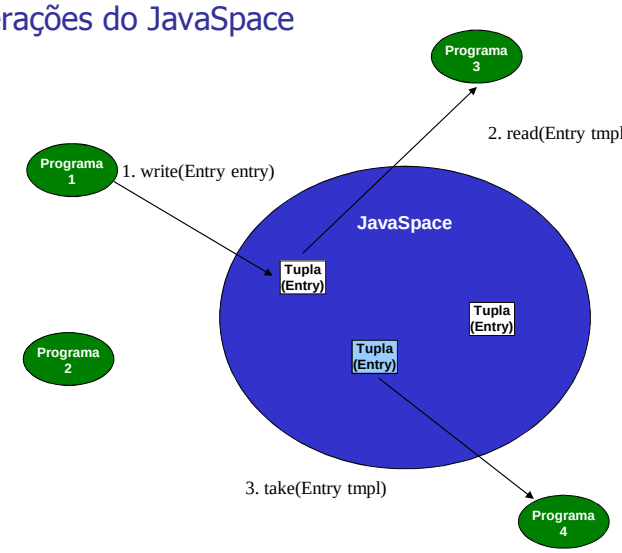


- Replicação Leitura-Escrita



Estudo de Caso: JavaSpace

- Operações do JavaSpace



Estudo de Caso: JavaSpace

- Métodos do JavaSpace

Method Summary	
EventRegistration	**notify**(Entry tmpl, Transaction txn, RemoteEventListener listener, long lease, MarshalledObject handback) When entries are written that match this template notify the given `listener` with a `RemoteEvent` that includes the `handback` object.
Entry	**read**(Entry tmpl, Transaction txn, long timeout) Read any matching entry from the space, blocking until one exists.
Entry	**readIfExists**(Entry tmpl, Transaction txn, long timeout) Read any matching entry from the space, returning `null` if there is currently is none.
Entry	**snapshot**(Entry e) The process of serializing an entry for transmission to a JavaSpaces service will be identical if the same entry is used twice.
Entry	**take**(Entry tmpl, Transaction txn, long timeout) Take a matching entry from the space, waiting until one exists.
Entry	**takeIfExists**(Entry tmpl, Transaction txn, long timeout) Take a matching entry from the space, returning `null` if there is currently is none.
Lease	**write**(Entry entry, Transaction txn, long lease) Write a new entry into the space.

Roteiro da aula

1. Acordo em sistemas distribuídos
2. Consenso em sistemas distribuídos
3. BFT

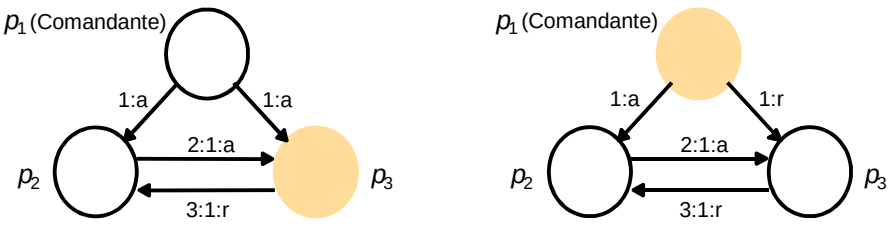
Acordo em sistemas distribuídos

- Exemplo de aplicação:
  - Computadores controlando os dispositivos de um navio decidir por proceder uma ação ou não;
  - Em transação bancária para transferir fundos de uma conta para outra, os computadores envolvidos numa transação devem consistentemente concordar por desempenhar o respectivo débito ou crédito;
  - Em exclusão mútua, os processos concordam em qual processo pode entrar na região crítica;
  - Em uma eleição, os processos concordam em qual é o processo a ser eleito;
  - Ordenação total em difusão atômica.

Problema dos generais bizantinos

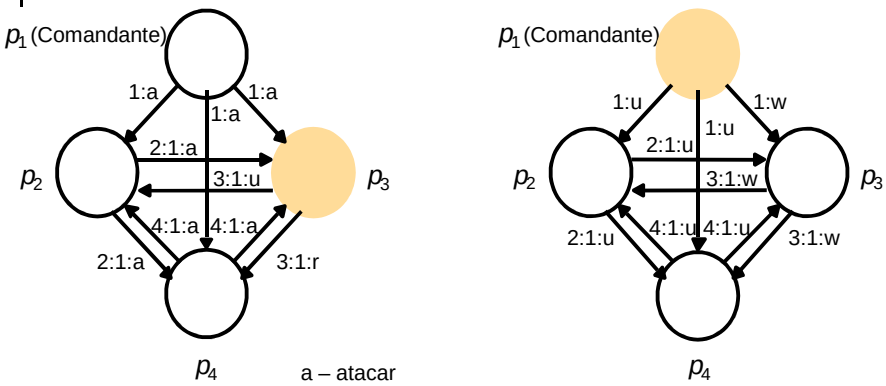
- Acordo bizantino
  - Os tenentes devem concordar em atacar ou retirar;
  - O general comandante edita uma ordem aos tenentes;
  - Os tenentes devem confirmar a ordem do general para decidir em atacar ou retirar;
- Propriedades:
  - Acordo: o valor de decisão de todos os processos corretos é o mesmo. Se Pi e Pj são corretos e tem entrado no estado decidido, Di = Dj (i, j = 1,2,...N);
  - Integridade: se o comandante é correto, então todos processos corretos decidem no valor que o comandante propôs;
  - Terminação: todos processos corretos terminam por tomar uma decisão.

Problema dos generais bizantinos



a – atacar
r – retirar
Processos faltosos estão sombreados

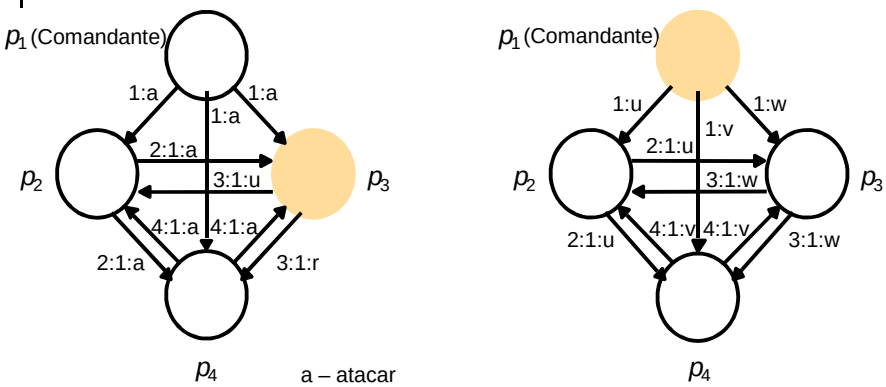
Problema dos generais bizantinos



a – atacar
r – retirar
Processos faltosos estão sombreados

N = 3f + 1
- N – número de processos no grupo
- f – número de processos faltosos (maliciosos)

Problema dos generais bizantinos



a – atacar
r – retirar
Processos faltosos estão sombreados

N = 3f + 1
- N – número de processos no grupo
- f – número de processos faltosos (maliciosos)