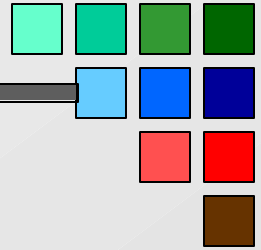
Sistemas Distribuídos

Ricardo Ribeiro dos Santos

ricrs@ec.ucdb.br

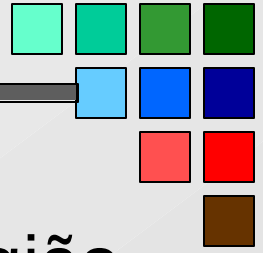
Curso de Engenharia de Computação – UCDB – Setembro/2003

Tópicos



- Sincronização em Sistemas Distribuídos
  - Exclusão Mútua
  - Transações Distribuídas

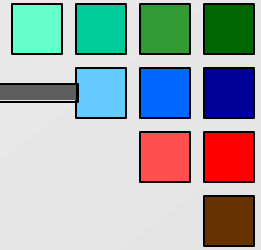
Exclusão Mútua



- Quando um processo deseja entrar em uma região crítica, ele procura atingir a exclusão mútua e garantir que nenhum outro processo utilizará informações compartilhadas ao mesmo tempo

- Métodos conhecidos para atingir exclusão mútua em SD:
  - Algoritmo centralizado
  - Algoritmo distribuído
  - *Token Ring*

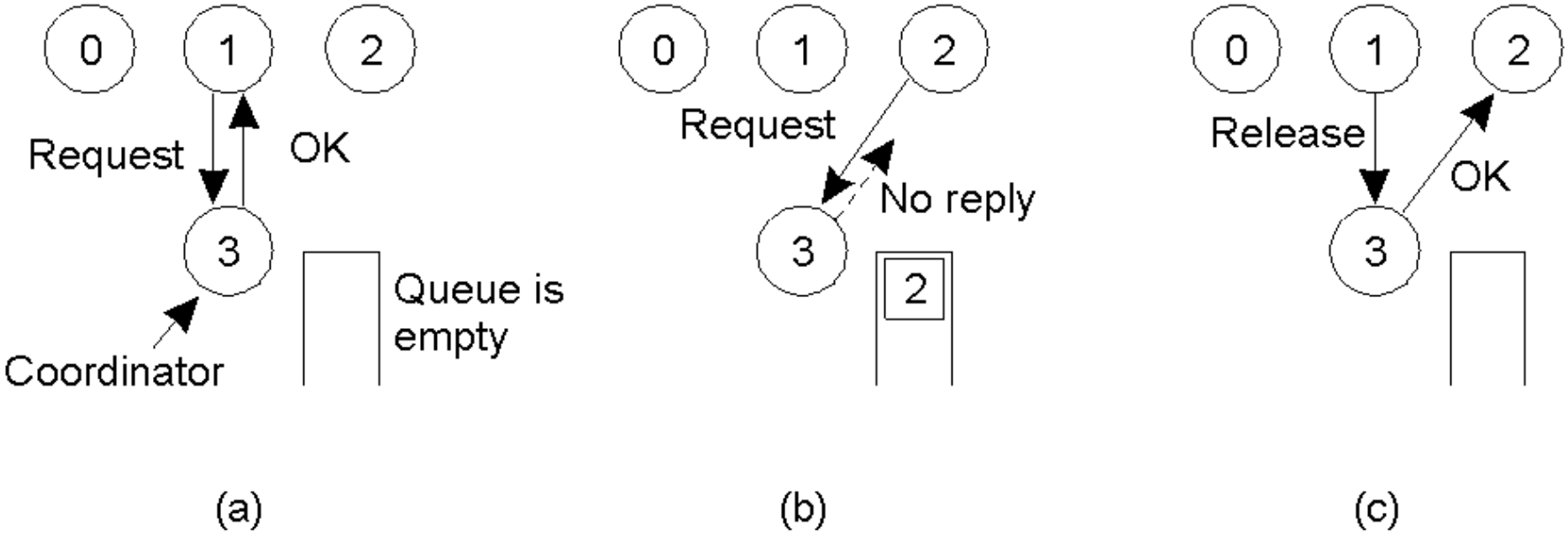
Exclusão Mútua



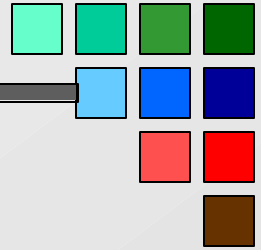
- Algoritmo centralizado
  - Elege-se um processo coordenador e sempre que alguém deseja entrar na região crítica, deve-se pedir permissão para o coordenador
  - Se nenhum outro processo estiver na região crítica, o coordenador concede permissão
  - Se algum processo já estiver na região crítica, a requisição é colocada em uma fila
  - Quando a região crítica é liberada por um processo, o coordenador concede permissão para a região crítica para quem estiver no começo da fila

Exclusão Mútua

- Algoritmo centralizado



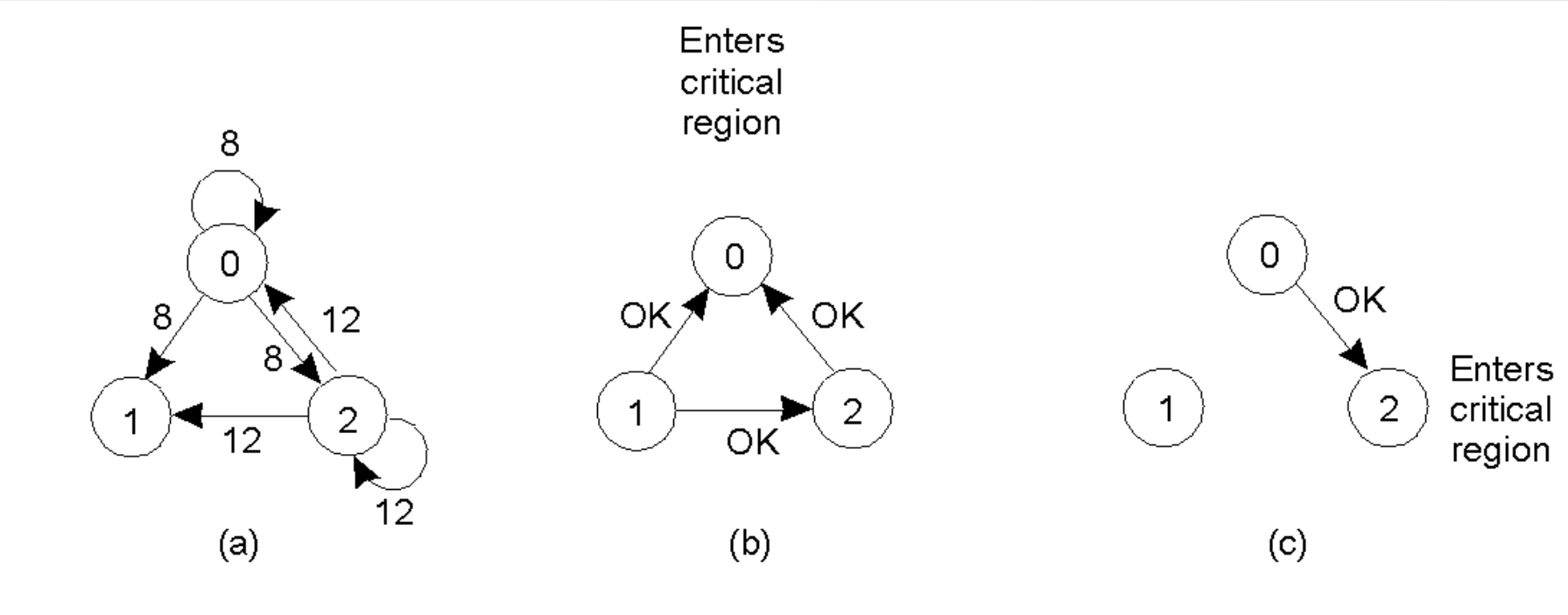
Exclusão Mútua



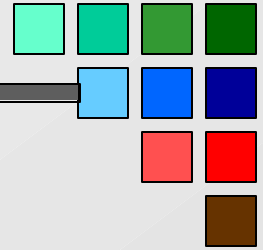
- Algoritmo distribuído (Ricart e Grawalla)
  - Assume que não deve existir ambigüidades na ordem dos eventos
  - Quando um processo deseja entrar em uma região crítica, ele informa o nome da região crítica, o seu número de processo e o tempo corrente (*timestamp*)
  - Essas informações são repassadas para todos os processos
  - Se um receptor não está na região crítica e não quer entrar nela, retorna um OK para o transmissor
  - Se o receptor estiver na região crítica, ele não responderá, ao invés, enfileira a solicitação
  - Se o receptor quiser entrar na região crítica, mas ainda não o fez, compara o *timestamp* da mensagem recebida com o *timestamp* da mensagem que ele mesmo enviou. A menor ganha

Exclusão Mútua

- Algoritmo distribuído



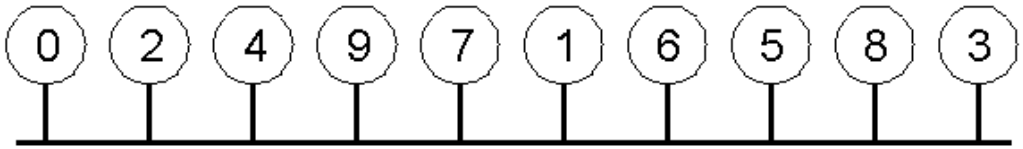
Exclusão Mútua



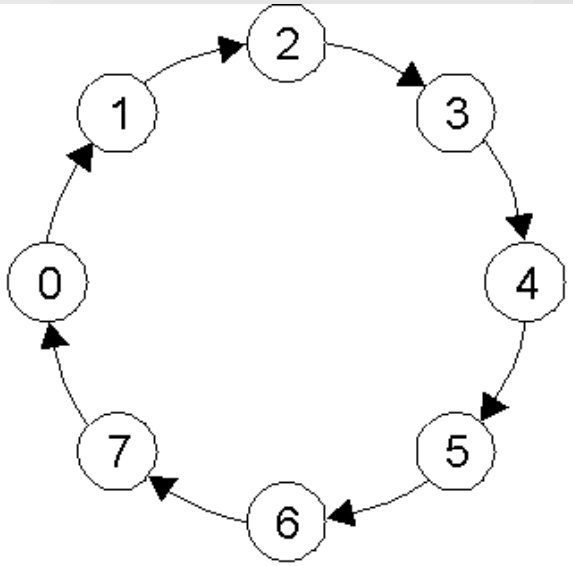
- Algoritmo *Token Ring*
  - Um anel lógico é definido no sistema
  - Cada processo é associado a uma posição do anel
  - Cada processo conhece seu sucessor
  - Há uma marca (*token*) que fica circulando pelo anel e é essa marca que dá direito a um processo acessar a região crítica
  - Quando um processo adquire a marca, ele checa para ver se precisa entrar na região crítica
  - Se precisar, entra na região, efetua o trabalho e sai da região
  - Depois de sair, passa a marca para o próximo processo do anel
  - Não é permitido entrar em uma 2ª região crítica usando a mesma marca

Exclusão Mútua

- Algoritmo *Token Ring*

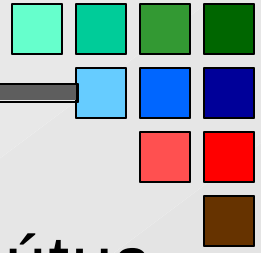


(a)



(b)

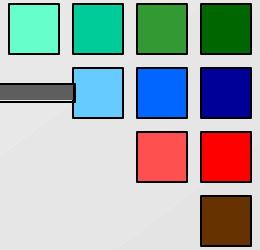
Exclusão Mútua



- Comparação entre os algoritmos de exclusão mútua

Algoritmo	**Mensagens por entrada/saída**	**Atraso antes de entrar**	**Problemas**
Centralizado	3	2	Coordenador
Distribuído	2 (n – 1)	2 (n – 1)	Qualquer processo
Token ring	1 até ∞	0 até n – 1	Perda do token

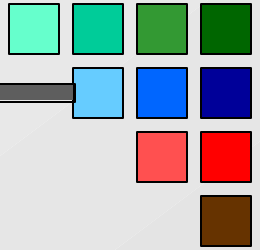
Transações distribuídas



- Transações são utilizadas para proteger dados compartilhados
- Um processo pode acessar e modificar múltiplos itens de dados como uma operação atômica

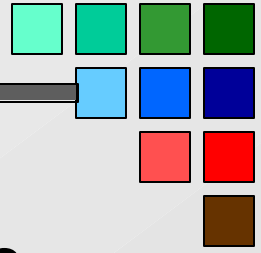
- Ex: usuário deseja debitar valor de uma conta e creditar esse valor em outra conta
  - Debitar (conta1, valor1)
  - Creditar (conta2, valor1)

Transações distribuídas



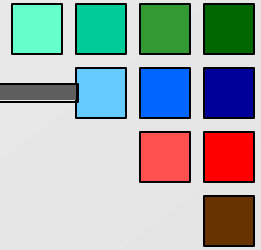
- Propriedades de Transações
  - Atômicas
    - Ocorre de forma indivisível
  - Consistentes
    - Não viola regras do sistema
  - Isoladas
    - Transações concorrentes não interferem umas nas outras
  - Duráveis
    - Uma vez realizado o *commit*, as mudanças são permanentes

Transações distribuídas



- Ex: usuário deseja debitar valor de uma conta e creditar esse valor em outra conta
  - Debitar (conta1, valor1)
  - Creditar (conta2, valor1)
- E se a conexão cair após a primeira ação (debitar) e antes da 2ª?
- Transações atômicas poderiam resolver problemas de inconsistência
  - Ou tudo será feito, ou nada o será!

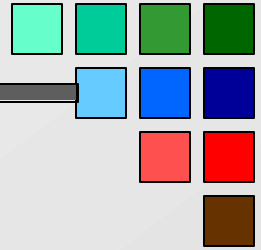
Transações distribuídas



- Problema da atualização perdida
  - A possui 100, B possui 200 e C possui 300

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

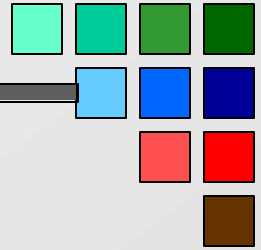
Transações distribuídas



- Problema da recuperação inconsistente
  - A e B possuem o valor 200

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

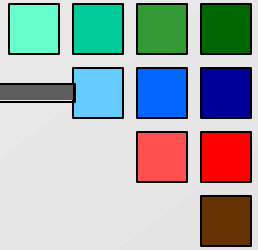
Transações distribuídas



- Resolução: Acesso serial

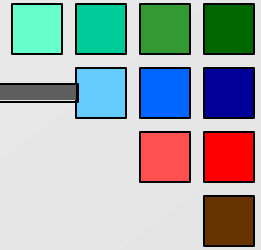
Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Transações distribuídas



- Primitivas de Transações
  - BEGIN_TRANSACTION
    - Início de uma operação
  - END_TRANSACTION
    - Termina a transação e tenta um commit
  - ABORT_TRANSACTION
    - Desfaz (rollback) todas as alterações
  - READ
    - Lê os dados de um arquivo
  - WRITE
    - Grava os dados para um arquivo

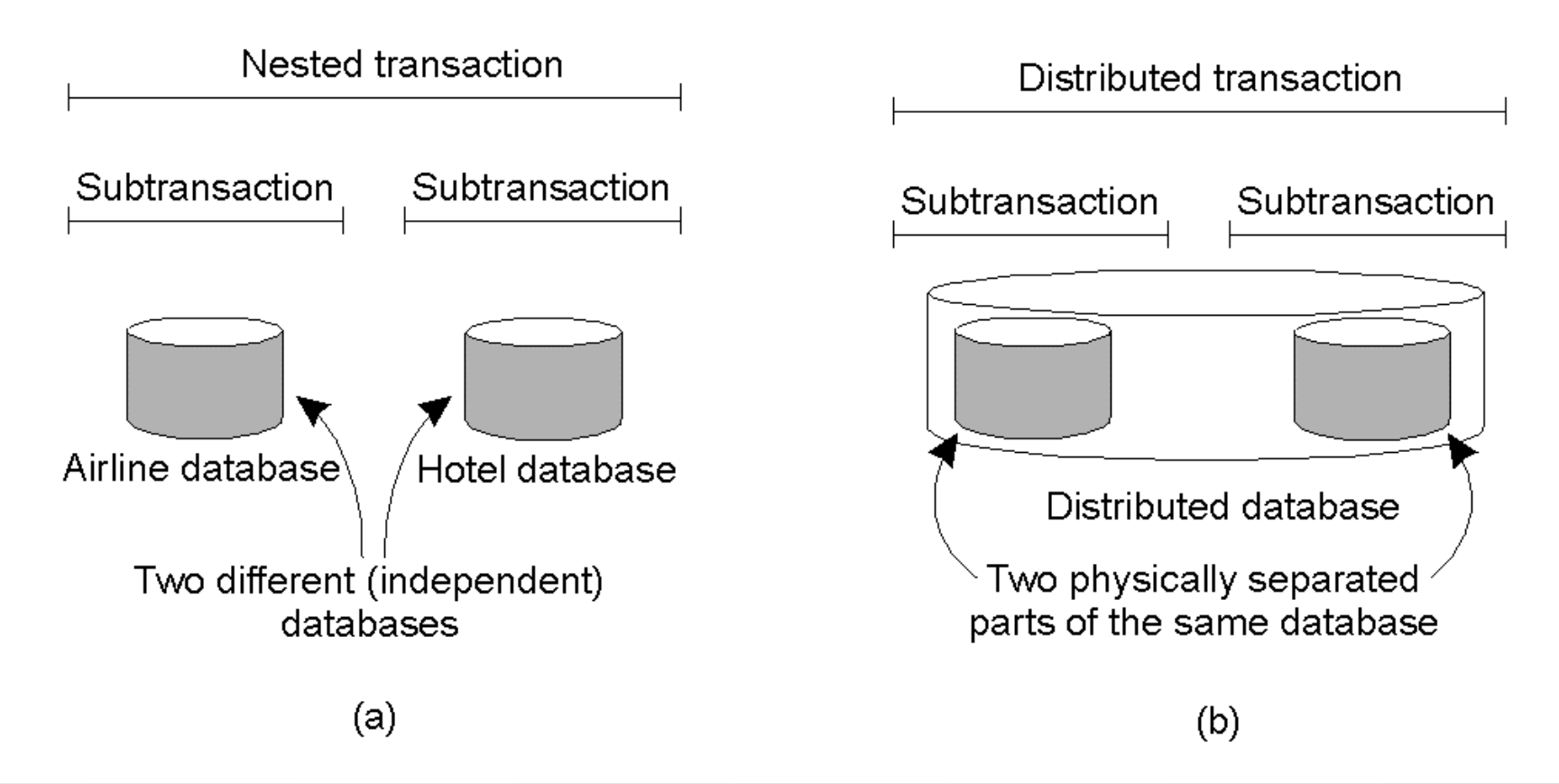
Transações distribuídas



- Classificação das transações:
  - *Flat*
    - Modelo habitual de transação. Consiste em uma unidade que realiza toda a operação ou desfaz toda a operação
  - Aninhada
    - Devido à característica da transação, pode-se dividir o trabalho em sub-transações visando a melhoria no desempenho
  - Distribuída
    - Em algumas situações, a transação deve operar de maneira distribuída
    - Por exemplo: uma transação que precisa gravar dados em BD distribuídos
  - Transações aninhadas são decompostas em uma hierarquia de sub-transações, enquanto que transações distribuídas são uma “espécie” de transação *flat* que opera sobre dados distribuídos

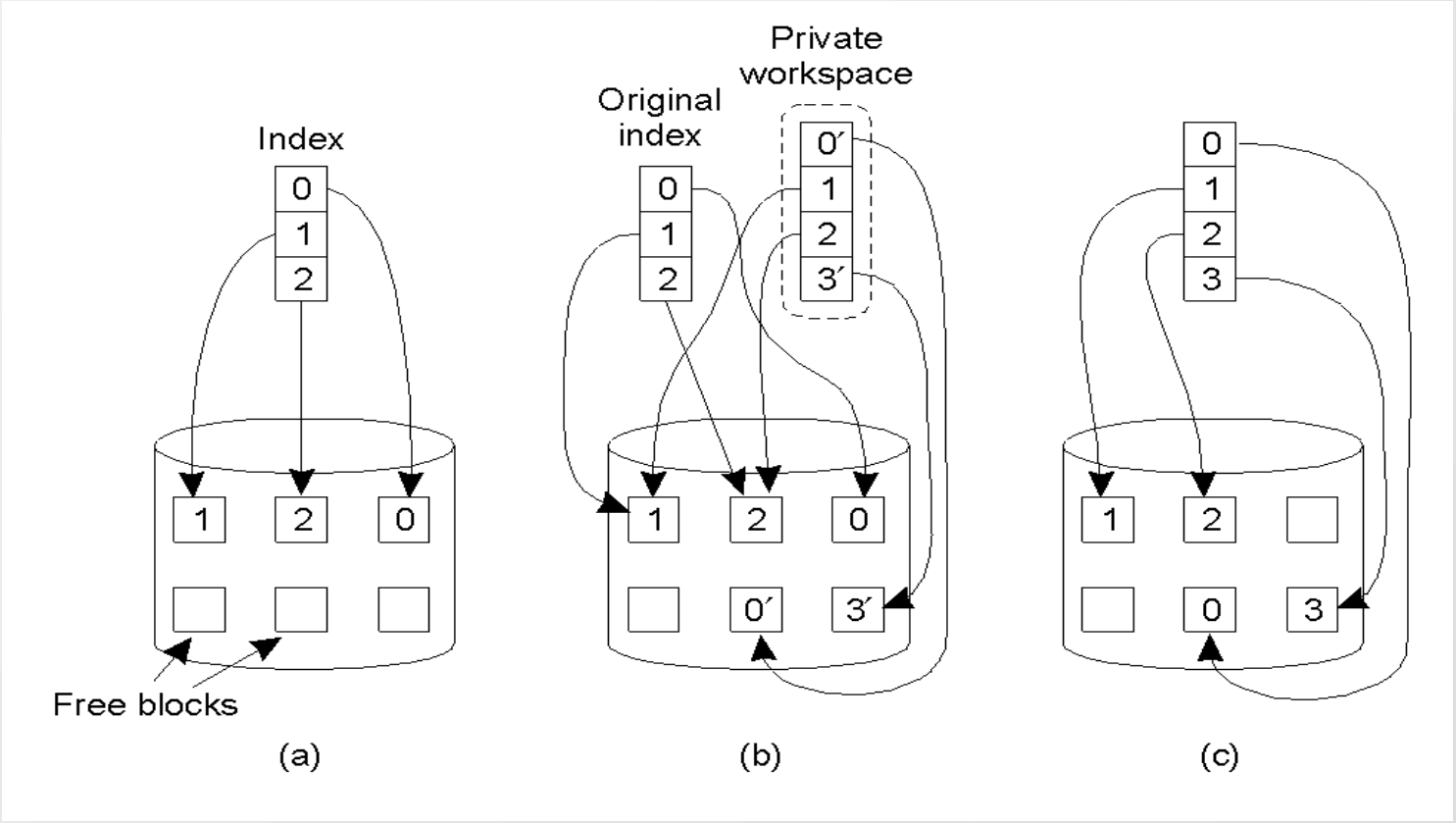
Transações distribuídas

- Classificação das transações:

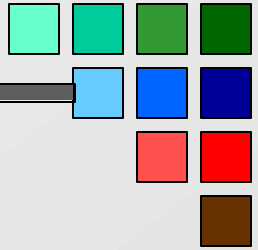


Transações distribuídas

- Implementação: Espaço de trabalho privado



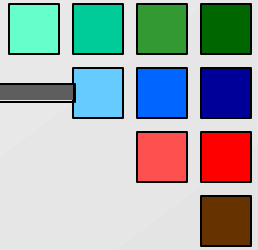
Transações distribuídas



- Implementação: Lista de intenções

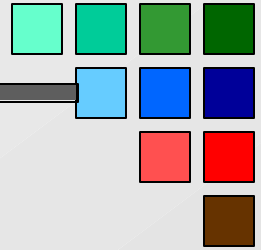
x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

Transações distribuídas



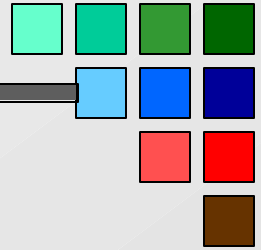
- Implementação: *Commit* de duas fases
  - Um dos processos funciona como coordenador
  - Coordenador escreve no log indicando que está inicializando o protocolo
  - Coordenador envia mensagem para os demais processos envolvidos indicando que preparem o *commit*
  - Subordinado verifica se já está realizando *commit*, escreve no log e envia sua decisão
  - Quando o coordenador recebe todas as respostas, ele decide se realiza o *commit* ou se aborta a transação

Transações distribuídas



- Controle de concorrência
  - Permitir que várias transações sejam executadas simultaneamente de maneira que os dados manipulados fiquem consistentes
  - O acesso das transações para o item de dados deve ser em uma ordem específica
  - Controle de concorrência pode ser determinado por três diferentes níveis:
    - gerente de dados;
    - escalonador;
    - gerente de transações

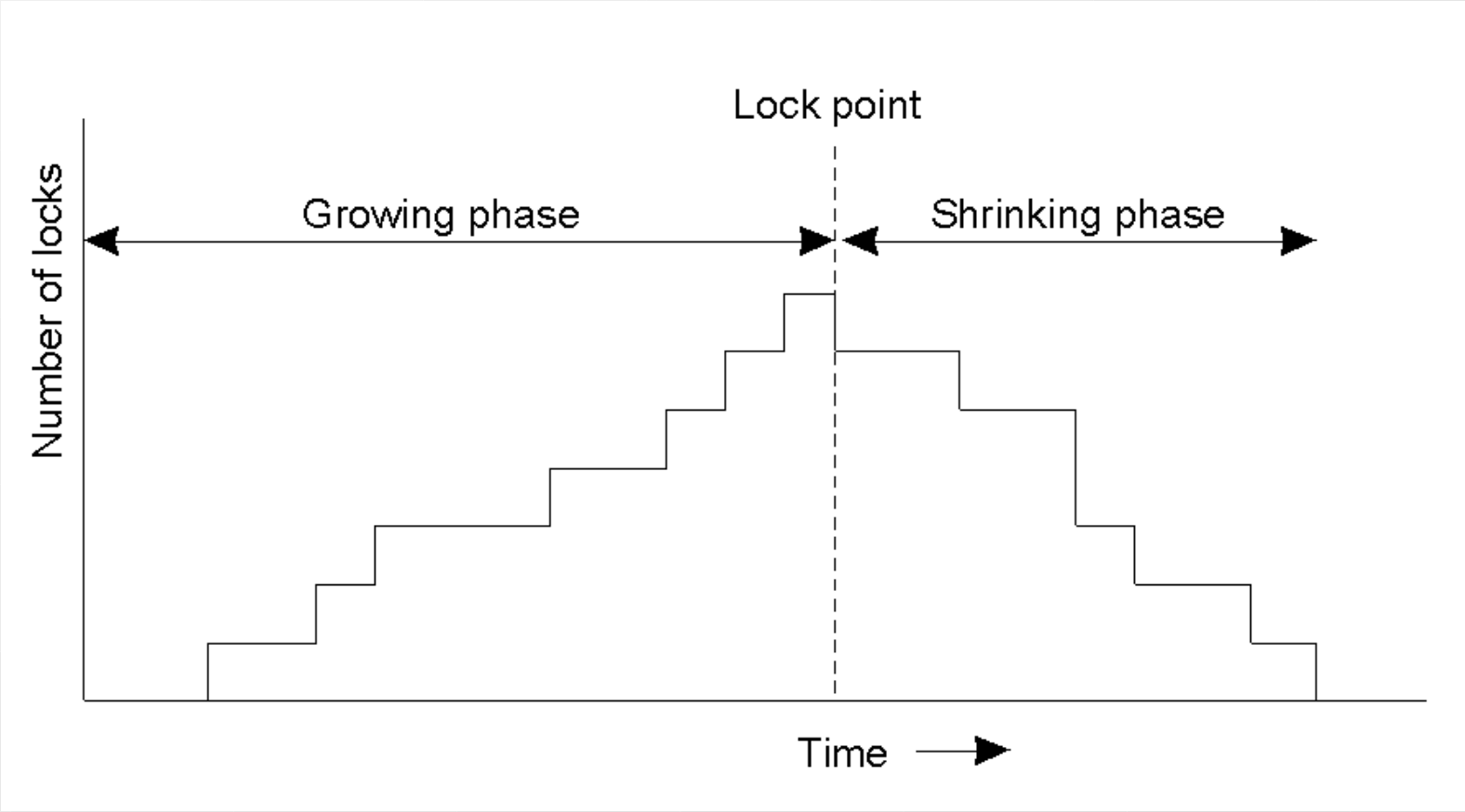
Transações distribuídas



- Controle de concorrência: Locks
  - “Travar” itens de dados para que sejam manipulados pela transação
  - Lock de duas fases:
    - Adquire todos os locks que necessita para realizar a operação
    - Realiza as operações
    - Libera os locks

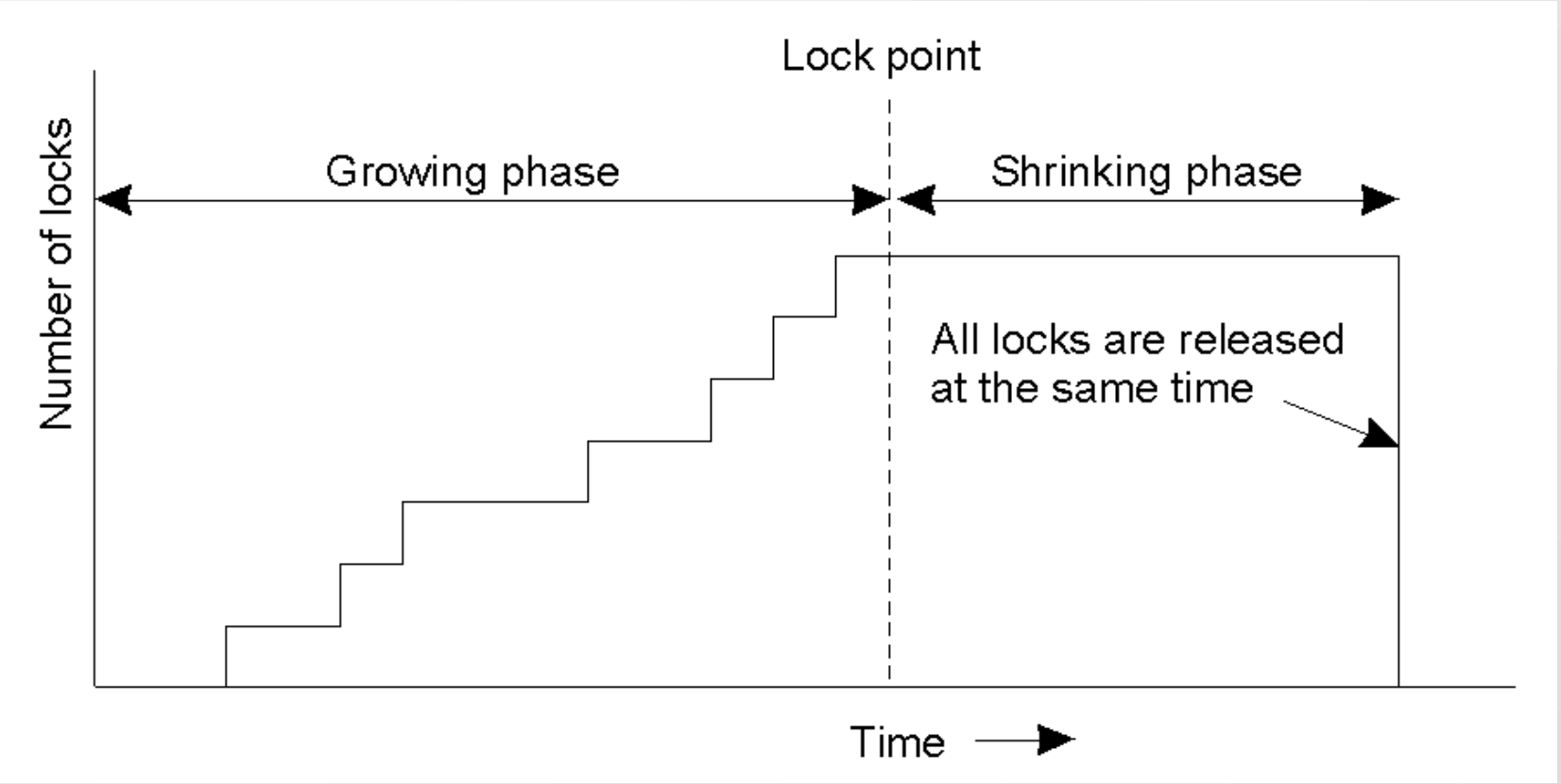
Transações distribuídas

- Locking de duas fases

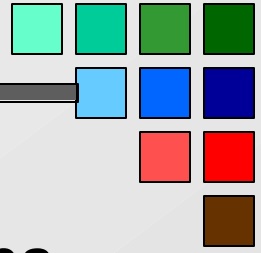


Transações distribuídas

- Strict Locking de duas fases



Transações distribuídas



- Controle de Concorrência: Ordem de *timestamps*
  - Abordagem pessimista:
    - Verificar inconsistências (uma transação com *timestamp* maior não deve alterar um item de dados antes de um transação com *timestamp* menor) antes de processar operações
  - Abordagem otimista
    - Executar operações até o final
    - Verificar se outras transações também modificaram os mesmos itens de dados
    - Em caso positivo, verificar se não há inconsistências de acordo com o *timestamp* das transações