Transações em Sistemas Distribuídos

Função transferir
Primeira solução

```
Mutex m[...];
transferir(contaA, contaB, montante)
{
   bancoA.lerSaldo(contaA, saldoA);
   bancoB.lerSaldo(contaB, saldoB);
   bancoA.atualizarSaldo(contaA, saldoA-montante);
   bancoB.atualizarSaldo(contaB, saldoB+montante);
}
```

um por cada conta

Quais os problemas desta solução?

Função transferir
Solução com Transação Atómica

```
transferir(contaA, contaB, montante)
{
   beginTransaction;
   bancoA.lerSaldo(contaA, saldoA);
   bancoB.lerSaldo(contaB, saldoB);
   bancoA.atualizarSaldo(contaA, saldoA-montante);
   bancoB.atualizarSaldo(contaB, saldoB+montante);
   commit;
}
```

Função transferir
Outra solução com Transação Atómica

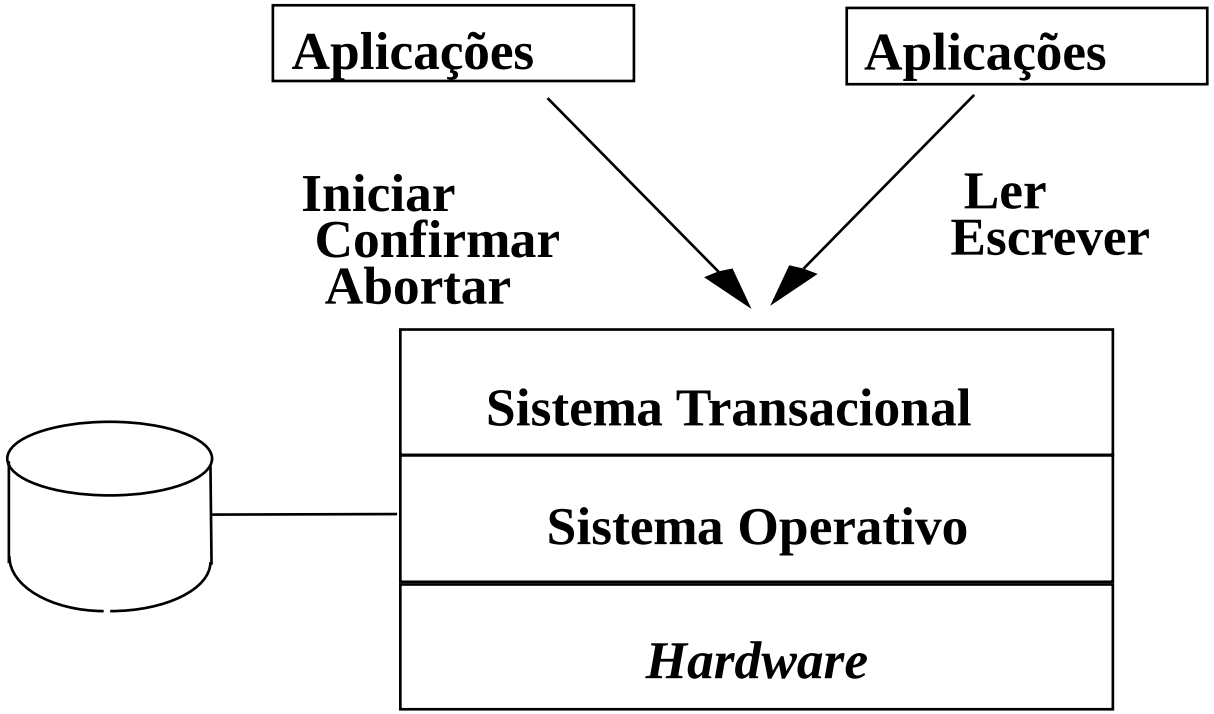
```
transferir(contaA, contaB, montante)
{
   beginTransaction;
   bancoA.lerSaldo(contaA, saldoA);
   bancoB.lerSaldo(contaB, saldoB);
   if (saldoA < montante) {
        abort;
   } else {
     bancoA.atualizarSaldo(contaA, saldoA-montante);
     bancoB.atualizarSaldo(contaB, saldoB+montante);
     commit;
   }
}
```

Em que situações pode a transação abortar?

Transações Atómicas Locais

(p)revisão da cadeira de Bases de Dados

Sistema Transacional



Transação

- Uma transação é uma sequência de leituras e escritas a objetos partilhados com outras transações

ACID

- Atomicidade
- Consistência
- Isolamento
- Durabilidade

Atomic, Consistent, Isolated, Durable

Propriedades das transações ACID
Atomicidade

- Transação ou se executa na totalidade ou não se executa
- O sistema tem de ser capaz de repor a situação inicial no caso da transação abortar (por iniciativa do programador ou falha do sistema)

Propriedades das transações ACID
Consistência

- Uma transação é uma transformação correta do estado
  - Supõe-se que o conjunto das ações da transação não viola nenhuma das regras de integridade associadas ao estado
  - Isto requer que a transação seja um programa correto

Propriedades das transações ACID
Isolamento

- Normalmente definida pela condição de **serializabilidade** (*serial-equivalence):*
  - Considere uma execução concorrente de leituras e escritas de múltiplas transações
  - A execução concorrente diz-se serializável quando existe uma execução sequencial (das mesmas transações) equivalente à execução concorrente
    - Ou seja, cujas leituras devolvam o mesmo valor e objetos escritos ficam com mesmo valor em ambas as execuções (concorrente e sequencial)

Propriedades das transações ACID
Isolamento

T1	T2
R(A)	
	R(B)
W(B)	
	W(A)
Com.	Com.

Esta execução concorrente é serializável?
Se sim, qual é a execução sequencial equivalente?

Propriedades das transações ACID
Isolamento

Transaction *T*:	Transaction *U*:
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Esta execução concorrente é serializável?
Se sim, qual é a execução sequencial equivalente?

Propriedades das transações ACID

- Durabilidade
  - Os resultados de uma transação que confirmou permanecem depois de esta acabar e sobrevivem ao conjunto de faltas expectáveis
  - Solução: resultados escritos em memória estável e com capacidade de recuperação das faltas dos discos que forem toleradas

Como gerir execuções concorrentes de transações?

Controlo de concorrência

- Solução pessimista (“pedir licença”)
  - Pressupõe que os conflitos são frequentes e obriga à prévia sincronização de todos os acessos

Trincos de Leitura/Escrita

	Leitura	Escrita
Leitura	Compatível	Conflito
Escrita	Conflito	Conflito

Controlo de concorrência pessimista
Sincronização em 2 fases estrita

- Cada objeto/grupo de objetos geridos por um trinco de leitura/escrita
- Sincronização em duas fases estrita (*strict two phase locking*):
  - À medida que a transação vai lendo/escrevendo sobre objetos, vai adquirindo sucessivamente os respetivos trincos (primeira fase)
  - Na terminação da transação (*commit* ou *abort*), liberta os trincos (segunda fase)

Controlo de concorrência pessimista
Sincronização em 2 fases estrita: exemplo

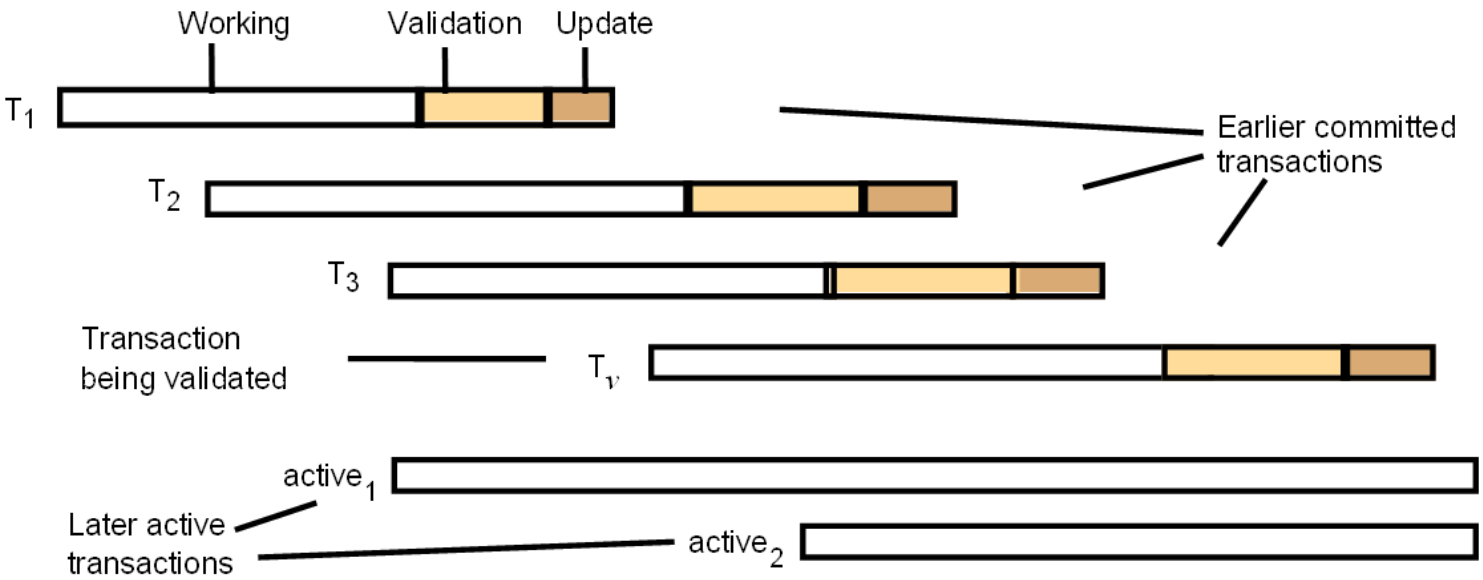
Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Isolamento (cont.)

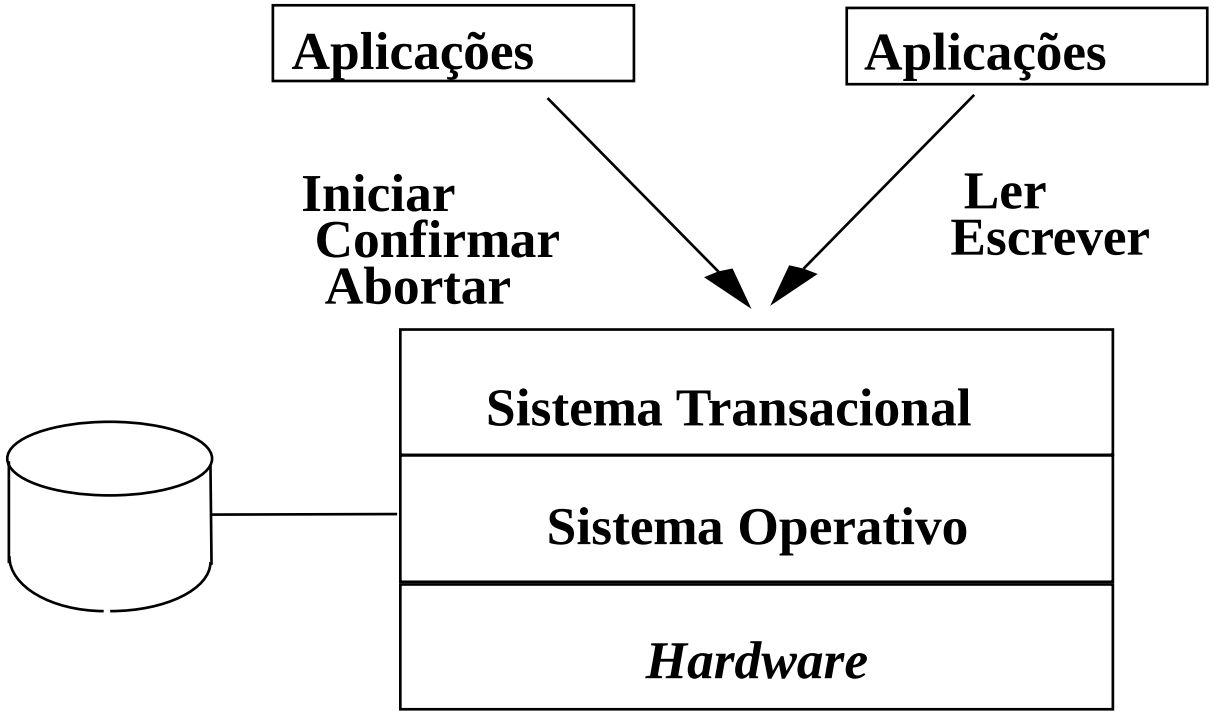
- **Solução pessimista (“pedir licença”)**
  - Pressupõe que os conflitos são frequentes e obriga à prévia sincronização de todos os acessos.
- **Solução otimista (“pedir desculpa”)**
  - Considera que os conflitos são raros
  - Na confirmação verifica a existência de conflitos
  - Obriga a manter carimbos temporais das atualizações para poder determinar quando há um conflito e nesse caso abortar as transações envolvidas

Solução otimista

T_v	T_i	Rule	
write	*read*	1.	T_i must not read objects written by T_v
read	*write*	2.	T_v must not read objects written by T_i
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i



Sistema Transacional



Transação

- Uma transação é uma sequência de leituras e escritas a objetos partilhados com outras transações

ACID

- Atomicidade
- Consistência
- Isolamento
- Durabilidade

Atomic, Consistent, Isolated, Durable

Transações Atómicas Distribuídas

Transações atómicas distribuídas

- Cliente pretende executar transação que envolve operações em múltiplos servidores
  - Os servidores chamam-se Participantes
  - Cada Participante é capaz de gerir transações locais
    - Por exemplo, através de base de dados local

- Como assegurar transação distribuída correta?
  - Em particular, atomicidade e isolamento

Transacções distribuídas: Modelo de Faltas

- Distribuição implica lidar com:
  - Falta dos discos
  - Falta das máquinas
  - Falta das comunicações
- Apenas se consideram faltas de paragem
  - Não se consideram faltas bizantinas
- Sistema assíncrono
  - Tempo de propagação das mensagens pode ser arbitrariamente longo
- Faltas temporárias de comunicação toleradas pelos protocolos de transporte
- Não se consideram faltas permanentes da rede (partições)

Suporte a transações locais

- Cada participante usa base de dados (ou outra fonte transacional) que suporta:
  - Começar transação local (*beginLocal*)
  - Executar escritas e leituras no âmbito de uma transação local ativa
  - Confirmar a transação (*commitLocal*) ou abortar (*abortLocal*)
- Muito importante:
  - Confirmar a transação nem sempre tem sucesso!
  - Razões para pedido de confirmar que resulta em transação abortada?

Como suportar transações distribuídas?

- Um processo coordena a transação distribuída
  - Por exemplo, um dos participantes
- Cliente começa por pedir um identificador da transação distribuída ao Coordenador
- Depois de iniciada a transação distribuída, cliente envia invocações diretamente aos Participantes
  - Cada pedido leva o identificador da transação distribuída
- Depois de correr a transação, o cliente pede ao Coordenador para a fechar

Interação Cliente-Coordenador
API do Coordenador

openTransaction() -> trans;

Inicia uma nova transação e atribui um TID único.
Este identificador é usado nas operações seguintes.

closeTransaction(trans) -> (commit, abort);

Termina a transação: um resultado *commit* indica que a transação foi confirmada; um resultado *abort* indique que foi abortada.

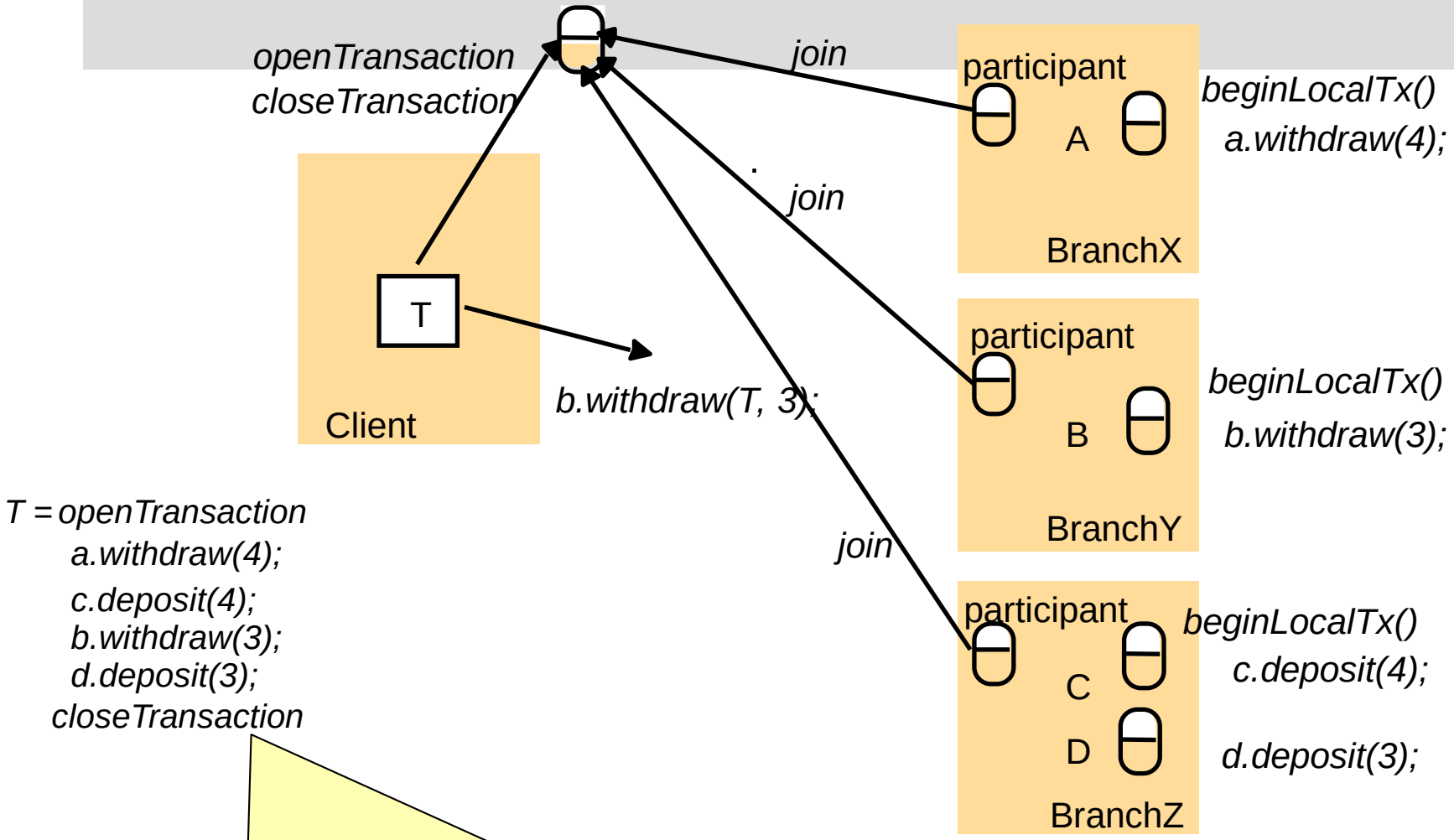
abortTransaction(trans);

Aborta a transação.

Papel de cada Participante

- Invocações a cada Participante levam o identificador da transação distribuída
- Ao receber cada pedido, o Participante:
  - Verifica se já participa na transação distribuída
  - Se não:
    - Inicia nova tx. local, que fica associada à tx. distribuída
    - Avisa Coordenador
  - Executa pedido na tx. local associada à tx. distribuída

Exemplo



O que acontece quando chegamos a closeTransaction?

E quando se chega ao *closeTransaction*?

- Cliente envia closeTransaction(id) ao Coordenador
- Coordenador envia ordem de *commit* a todos os Participantes que fizeram *join* à transação distribuída

Boa solução?

Protocolo de confirmação em duas fases

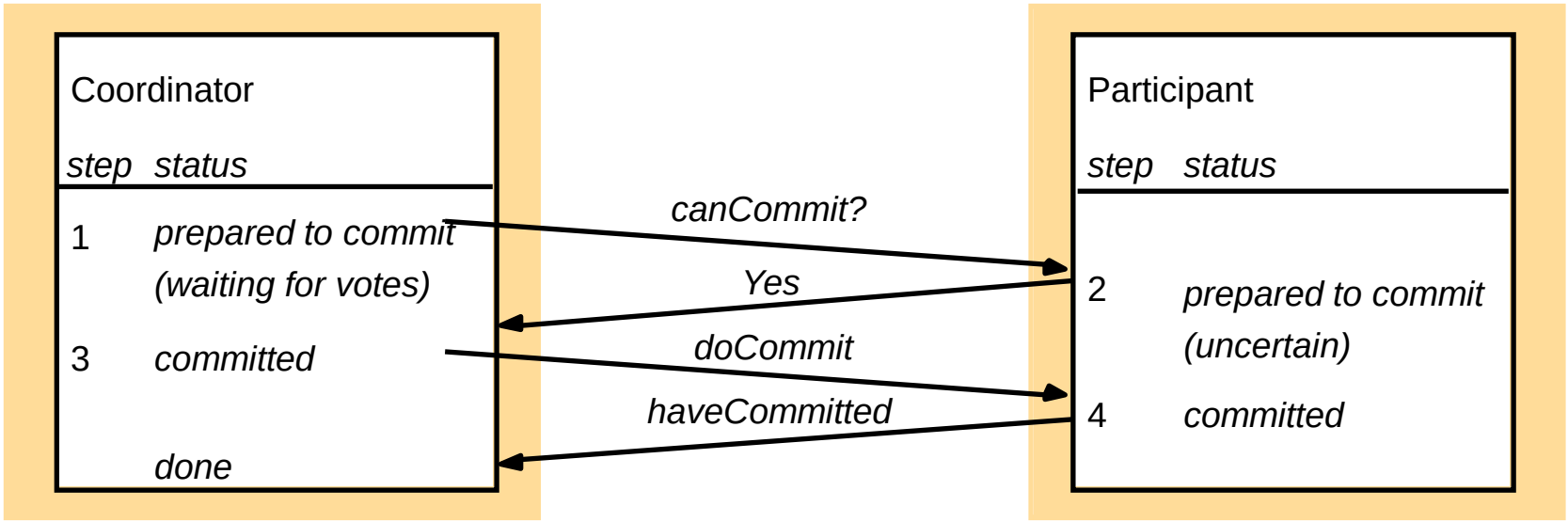
Two-Phase Commit (2PC)

Transações distribuídas: Problemas a considerar

- A tomada de decisão de abortar ou confirmar uma transacção é o problema mais complexo a resolver

- Requer um consenso entre os diferentes participantes numa transação distribuída

Protocolo (sem falhas)



Protocolo (sem falhas)

Coordenador

Envia canCommit a todos

if (todos votaram sim)
    decisão$_{coord}$ = *commit*
    Envia doCommit a todos
else
    decisão$_{coord}$ = *abort*
    Envia doAbort a todos que votaram sim
exit

Participante i

Envia voto$_i$ ao coord.
if (*voto*$_i$ == *não*)
    decisão$_i$ = *abort*
    exit

if (recebe ABORT)
    decisão$_i$ = *abort*
else
    decisão$_i$ = *commit*
exit

Interação coordenador-participante

canCommit?(trans)-> Yes / No

Chamada do Coordenador para o Participante para perguntar se quer confirmar a transação. O Participante responde com o seu voto.

doCommit(trans)

Chamada do Coordenador para o Participante para dizer ao Participante que deve confirmar a sua parte da transação.

doAbort(trans)

Chamada do Coordenador para o Participante para dizer ao Participante que deve abortar a sua parte da transação.

haveCommitted(trans, participant)

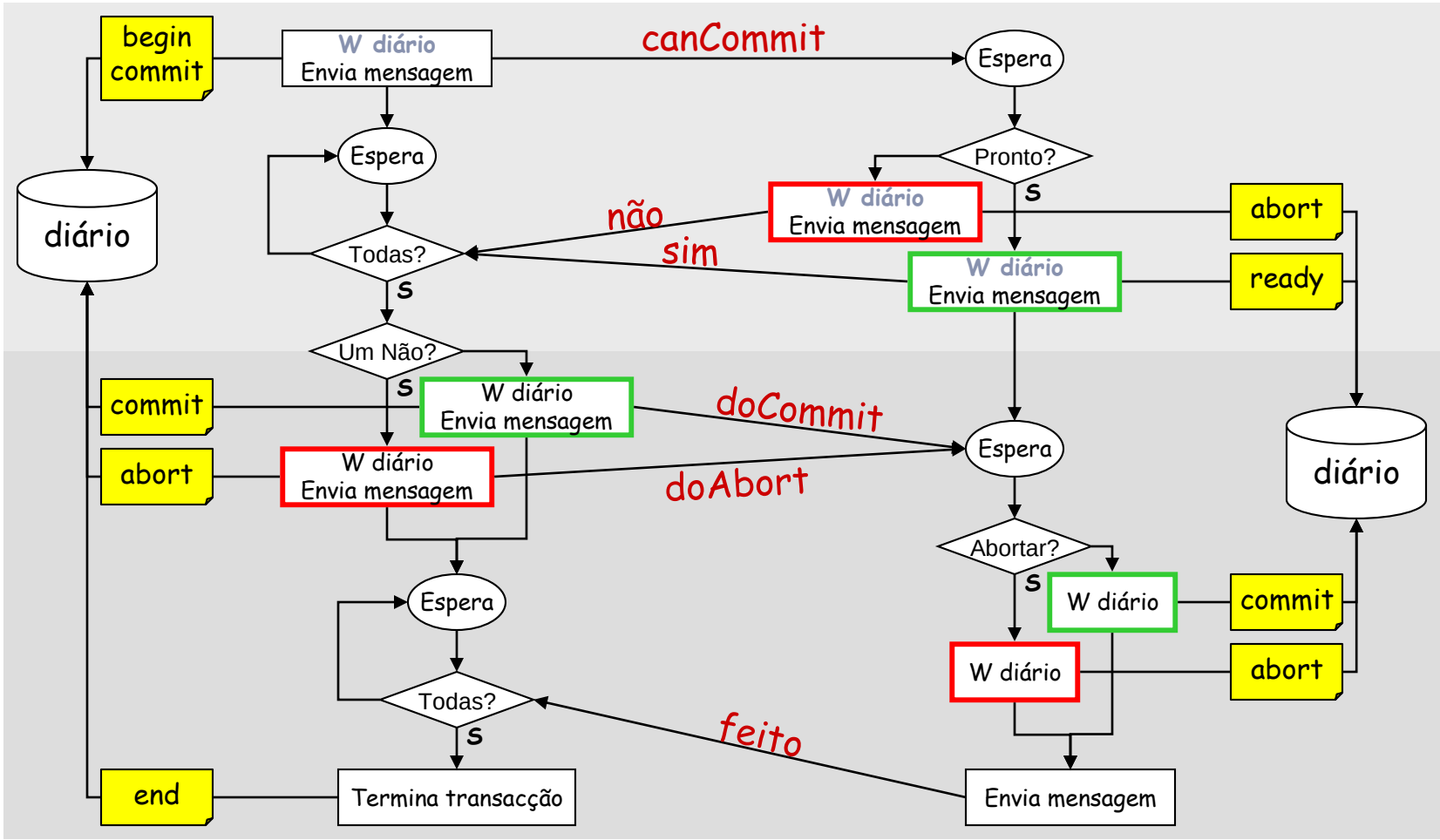
Chamada do Participante para o Coordenador para informar que confirmou a transação.

getDecision(trans) -> Yes / No

Chamada do Participante para o Coordenador para pedir a decisão sobre a transação depois de ter votado Yes mas sem ter ainda recebido resposta, ao fim de um tempo. Usado para recuperar de uma falha do servidor ou de mensagens em atraso.

E caso haja falhas de processos ou atrasos na rede?

A necessidade de registar as decisões num diário persistente



Coordenador **Participante**

Logs do protocolo

Evento	Info. escrita no *log*	Instante da escrita
Coord. envia canCommit	Begin *commit*	Em paralelo com envio
Participante vota sim	Sim	Antes de enviar voto
Participante vota não	Não	Em paralelo com envio
Coord. decide *commit*	*Commit*	Antes de enviar *commit*
Coord. decide *abort*	*Abort*	Em paralelo com envio
Particip. recebe decisão	*Commit* ou *Abort*	Em paralelo com decisão

Diagrama de Estados - Coordenador

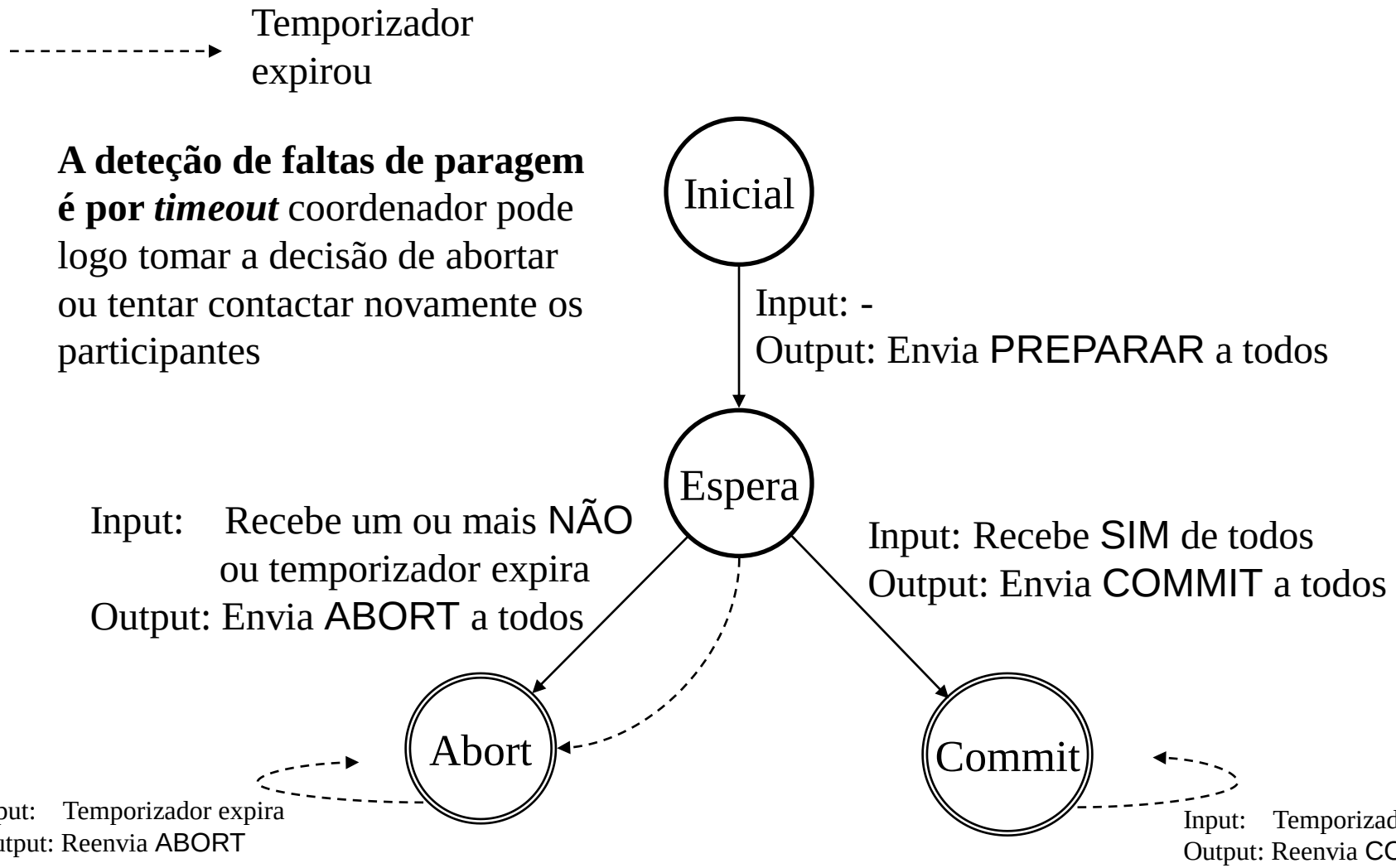
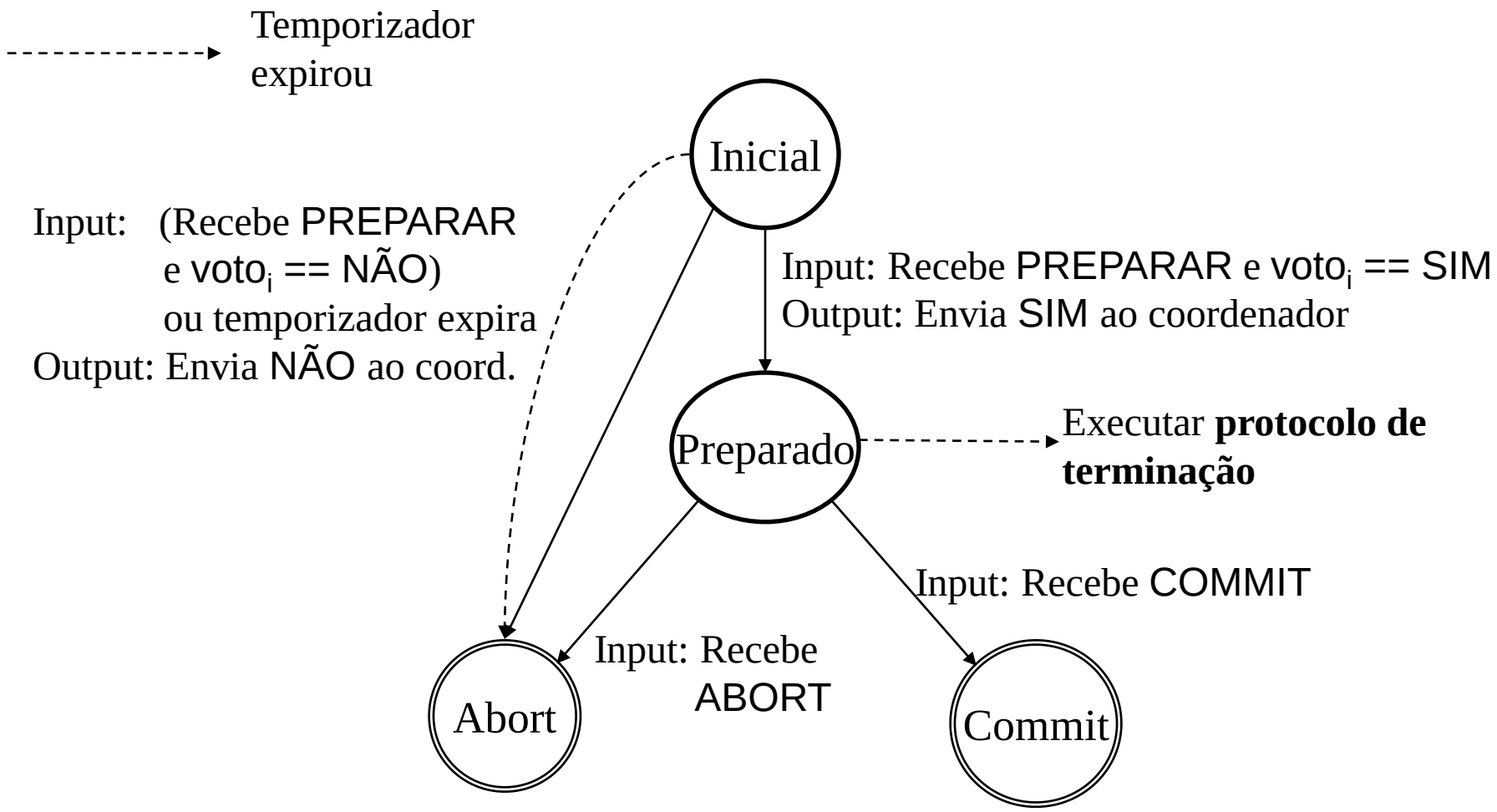


Diagrama de Estado - Participante



Transações distribuídas: Tolerância a faltas no 2PC

- Não receção de mensagens
  - Detetadas com um temporizador no Coordenador ou nos Participantes
- *Timeout* no Coordenador
  - Estado Esperar
    - **Não pode confirmar unilateralmente a transação**
    - Mas pode unilateralmente optar por abortar a transação
      - Se considerar que o atraso na resposta se deve a uma falta
  - Estados Abortar e Confirmar
    - **O Coordenador não pode terminar a transação**
      - Tem que receber a confirmação de todos os Participantes
    - Pode repetir a mensagem global previamente enviada

Transações distribuídas: Tolerância a faltas no 2PC

- *Timeout* num Participante
  - Estado Inicial
    - Pode optar unilateralmente por abortar a transação
    - Ou verifica o estado do Coordenador
  - Estado Preparado
    - Não pode progredir
      - Depende da decisão do Coordenador que já influenciou
      - A transação fica ativa e bloqueada até se saber essa decisão
    - Se os Participantes interatuarem é possível evoluir
      - » Obtendo a decisão do Coordenador que chegou aos outros Participantes

Recuperação depois de falta de paragem

- Recuperação do Coordenador
  - Estados Inicial e Esperar
    - Repete as mensagens de Preparação para obter novamente a votação dos participantes
  - Estado Confirmar ou Abortar
    - Se ainda não recebeu todas as confirmações repete o envio da mensagem global previamente enviada
- Recuperação de um Participante
  - Estado Inicial
    - Aborta unilateralmente a transação
  - Estado Preparado
    - Reenvia o seu voto (sim ou não) para o Coordenador

Transações distribuídas: Problemas do 2PC

- O protocolo é bloqueante:
  - Obriga os Participantes a esperar pela recuperação do Coordenador
  - E vice-versa
- Não é possível fazer uma recuperação totalmente independente
- Há alternativas não-bloqueantes
  - Sob modelos de faltas mais restritivos
  - Normalmente muito mais complexas

Transacções distribuídas

Arquitectura X/Open

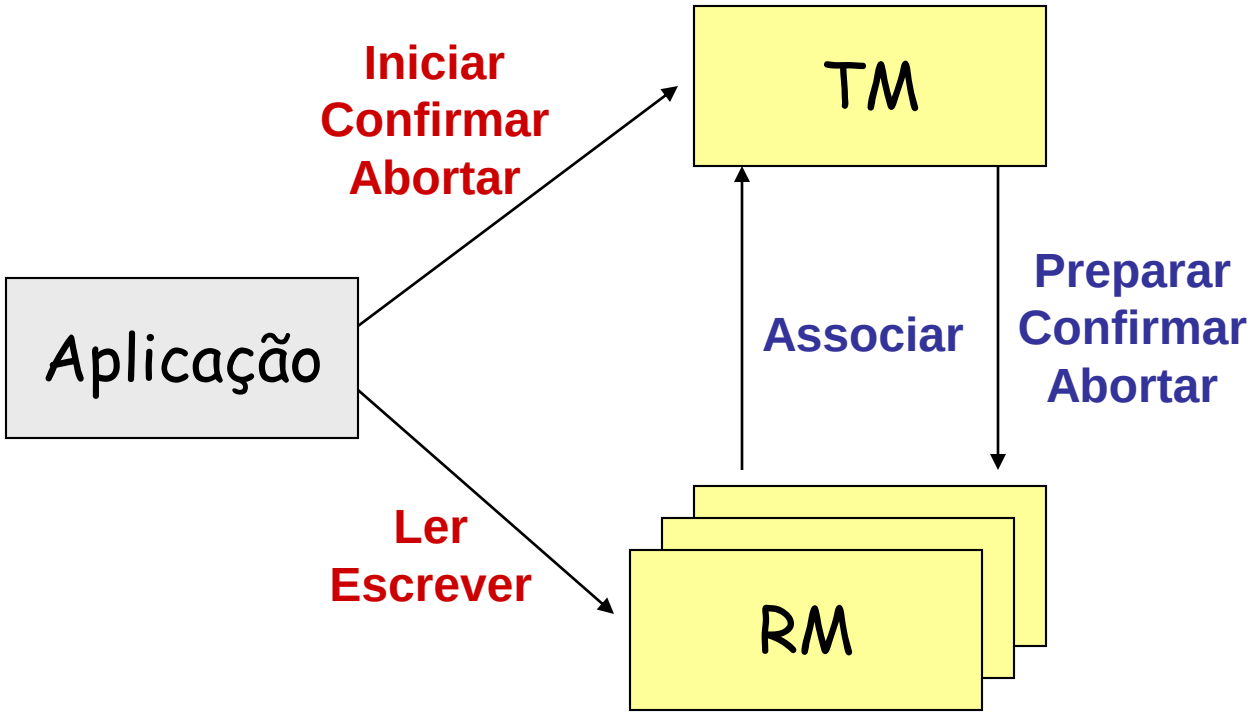
Consórcio X/OPEN

- Esforço de normalização dos protocolos e interfaces para interligação de sistemas de informação heterogéneos
  - DTP (Distributed Transaction Processing)
  - Muito influenciado pela norma de facto que constituiu a arquitectura SNA da IBM e a sua interface LU 6.2

Arquitectura X/Open

- Gestores de Recursos - RM (resource manager)
  - Armazenam os dados
    - Em BDs relacionais, sistemas de ficheiros com actualizações atómicas, etc.
  - Garantem localmente as propriedades das transacções
- Monitores Transaccionais - TM (transaction managers)
  - Coordenadores dos RM
    - Através da interface XA
  - Execução dos protocolos de iniciação/terminação das transacções
  - Um em cada máquina (ou em cada grupo de máquinas)

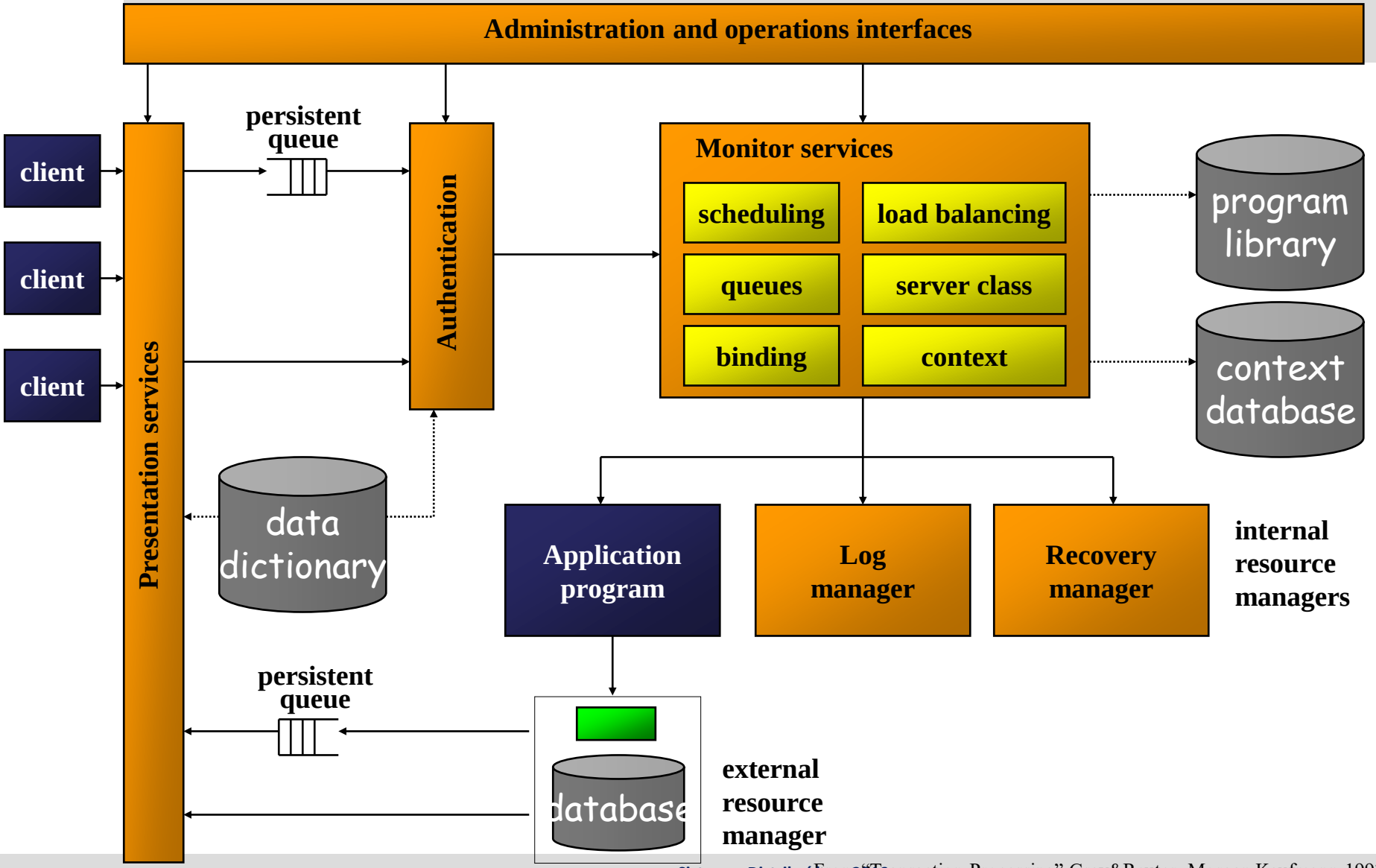
Transações distribuídas X/Open



Transações distribuídas: X/Open

- Uma aplicação inicia uma transação
  - Invoca o TM local que lhe atribui um identificador único
- A aplicação contacta em seguida os RMs
  - Para efetuar as operações de leitura e escrita
  - Usa o identificador da transação
- Os RMs associam-se à transação
  - Quando um RM recebe a primeira operação relacionada com uma transação desconhecida contacta o TM local para se associar à transação
- A transação termina
  - Todos os TM envolvidos executam um protocolo de consenso distribuído

TP-Monitor components (generic)



From "Transaction Processing" Gray&Reuter. Morgan Kaufmann 1993

Monitor Transaccional Tuxedo da BEA