Replication in Databases and Distributed Systems, and Parallel Programming with Transactional Memory

Alexandre Oliveira m2066

Marco Oliveira m1511

Paulo Malheiro a14679

Lista de conteúdos

- Introdução;
- Contexto da Replicação;
- Modelo funcional;
- Replicação em Sistemas Distribuídos;
- Replicação em Bases de Dados;
- Problemas de Sincronização;
- O Dilema do Programador;
- Memória Transaccional;
- Conclusões;
- Bibliografia.

Introdução:

- Estudo da replicação em diversas áreas:
  - Em sistemas distribuídos, o estudo da replicação é feito para efeitos de tolerância a falhas;
  - Em bases de dados, a replicação é vista como forma de melhorar o desempenho e disponibilidade das mesmas.
- As técnicas e mecanismos criados nestas 2 áreas, são conceptualmente idênticos mas tal não se verifica em termos práticos;
- A velocidade dos processadores não tem aumentado ao nível verificado nas últimas décadas.
  - A necessidade de maior velocidade é real.
- As cada vez mais complexas aplicações levam os programadores a procurar alternativas.
  - Surgem os processadores de vários *cores*.
- É necessário paralelizar!
  - Múltiplos caminhos de execução terão de cooperar para completar tarefas do programa, e parte acontecerá de forma concorrente.

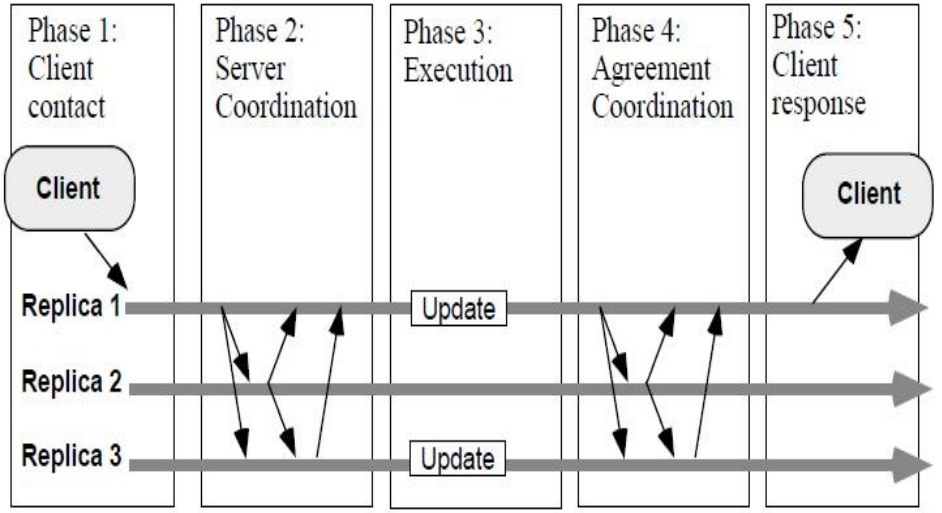
Contexto da replicação:

- De uma forma geral o sistema é formado por um conjunto de réplicas, sobre as quais são executadas operações pedidas por clientes, e cuja comunicação é feita através de troca de mensagens;
- Assim sendo os sistemas distribuídos podem ser classificados como síncronos e assíncronos:
  - **Síncronos:** limites para a velocidade de processamento e atrasos na transmissão das mensagens; permitem a detecção correcta de avarias.
  - **Detecção correcta de avarias:** processo ***p*** pensa que um outro processo ***q*** "crashou", e esse "crash" verifica-se no sistema.
  - **Assíncronos:** não existem limites para a velocidade de processamento ou atrasos na comunicação; não permitem a detecção correcta de avarias.
- As bases de dados não são caracterizadas da mesma forma que os sistemas distribuídos, pois "vivem" de acordo com protocolos de bloqueio ao passo que os sistemas distribuídos usam protocolos que evitam o bloqueio.
  - **Protocolo de bloqueio:** protocolo que bloqueia se a falha de alguns dos processos envolvidos pode prevenir que este termine a sua execução.

Modelo funcional

- Qualquer mecanismo ou técnica de replicação pode ser descrito em 5 fases genéricas:[1]

  1. **Request (RE);**
  2. **Server Coordination (SC);**
  3. **Execution (EX);**
  4. **Agreement Coordination (AC);**
  5. **Client Response (END).**



Posteriormente vamos verificar que algumas técnicas de replicação podem:

- Não considerar algumas das fases;
- Ordena-las de forma diferente;
- Juntar algumas delas numa sequência mais simples.

- Assim sendo estas técnicas são comparadas pela forma como implementam cada uma das fases, e como as combinam para criar uma sequência mais simples.

Modelo funcional (cont.)

Request (RE):

- O cliente envia um pedido (operação) ao sistema:
- Directamente a todos as réplicas existentes no sistema;
- Envia a uma réplica que, por sua vez, se encarregará de retransmitir às restantes réplicas como parte da fase **SC** (**S**erver **C**oordination).
- Esta distinção introduz imediatamente algumas diferenças significativas entre sistemas distribuídos e bases de dados:
- Em bases de dados nunca contactam todas as réplicas do sistema, enviando o pedido a somente uma delas. Porquê?
    - Porque a replicação deve ser transparente para o cliente e...
    - Ser capaz de enviar um pedido a todas as réplicas implica o conhecimento, por parte do cliente, da localização da informação, esquemas e distribuição. Algo não muito prático para bases de dados de tamanho considerável.
- Em sistemas distribuídos, no entanto, existe uma distinção entre as diversas técnicas de replicação dependendo se o cliente envia o pedido a todas as cópias (**replicação activa**) ou somente a uma (**replicação passiva**).

Modelo funcional (cont.)

Server Coordination (SC):

- As diferentes réplicas tentam encontrar uma ordem pela qual os pedidos/operações necessitam ser executadas.
- É neste ponto que as técnicas diferem mais em termos de estratégias e mecanismos de ordenação, bem como critérios de correcção.
- As bases de dados ordenam estas operações de acordo com as dependências dos dados, pois todas as operações têm que possuir as mesmas dependências em todas as réplicas e usam a serialização adaptada a cada situação como critério de correcção.
- Os sistemas distribuídos, por seu lado, usam linearização e consistência sequencial:
- A linearização é baseada em dependências (em tempo real) enquanto que a consistência sequencial apenas considera a ordem pela qual as operações são executadas, por cada processo.

Modelo funcional (cont.)

Execution (EX):

- Esta representa a actual execução da operação.
- Não introduz muitas diferenças entre abordagens/técnicas existentes, sendo um bom indicador de como cada uma dessas abordagens, trata da distribuição das operações e como lida com as ameaças ao sistema.
- A aplicação das actualizações, resultantes da aplicação das operações, é normalmente feita na fase seguinte (**Agreement Coordination**).

Modelo funcional (cont.)

Agreement Coordination (AC):

- Nesta fase todas as réplicas asseguram-se que estão a realizar as mesmas operações.
- Esta fase é considerada a mais interessante de todo o modelo funcional pois é nesta que se evidenciam as diferenças fundamentais entre as diversas técnicas/protocolos.
  - Em bases de dados, esta fase é necessária pois a anterior (**SC**) realiza apenas a ordenação das operações. Após tal operação, todas as réplicas devem concordar entre si essa sequência de operações.
  - De notar que a ordenação das operações não significa que estas irão ser bem sucedidas na sua execução. Existem diversos factores em bases de dados, que podem levar a que uma sequência não seja bem sucedida numa réplica (**carga, limitações na consistência, interacções com operações locais**).
  - É neste último ponto que encontramos a principal diferença entre SD e BD, pois nos primeiros a partir do momento em que uma sequência de operações é totalmente ordenada, esta irá ser entregue (executada).

Modelo funcional (cont.)

Client Response (END):

- Esta fase representa o momento em que o cliente recebe a resposta do sistema, sendo que existem 2 resultados possíveis para esta fase:
  - A resposta é enviada após tudo estar acordado e a sequência de operações ter sido executada ou;
  - A resposta é enviada de imediato e tanto a propagação das mudanças feitas, pela execução das operações, como a coordenação entre todas as réplicas envolvidas é feita depois.
- No caso das bases de dados esta distinção leva ao estudo dos 2 tipos de técnicas existentes para esta área (eager e lazy replication).
- Em sistemas distribuídos a resposta é enviada somente após a execução do mecanismo/técnica e não tenham sido detectadas descrepâncias.

Replicação em Sistemas Distruibuídos

- Um sistema distribuído é modelado como um conjunto de serviços, implementados por processos servidor e invocados por processos clientes;
- Cada processo servidor possui um estado local, que é alterado de acordo com as invocações, e de uma forma atómica (modificações resultantes não são aplicadas de forma parcial);
- O isolamento entre invocações concorrentes é da responsabilidade do servidor, e é obtido através de mecanismos locais de sincronização;
- A tolerância a falhas é obtida por meio de serviços, e implementados por múltiplos processos servidor ou réplicas.

Por forma a encobrir a complexidade da replicação, foram introduzidas as noções de grupo (de servidores) e primitivas de comunicação em grupo.

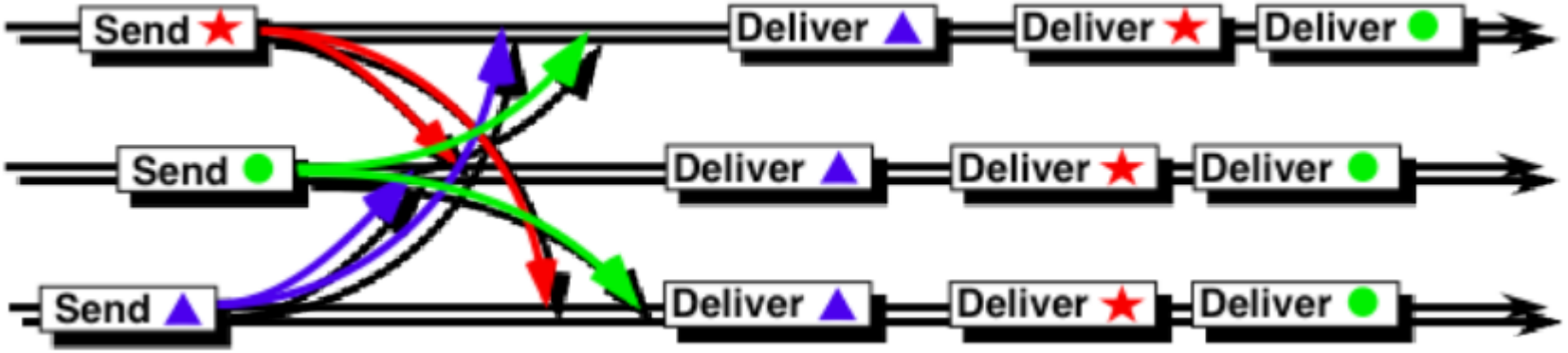
- A noção de grupo age como um mecanismo de endereçamento lógico, permitindo ao cliente ignorar o grau da replicação e a identidade de cada processo servidor, no serviço replicado.
- As primitivas de comunicação em grupo fornecem uma comunicação 1-n são 2: ***Atomic Broadcast*** e o ***View Synchronous Broadcast***

Atomic Broadcast:[3],[5]

- Temos 2 primitivas dentro do ABCAST/ TO-CAST:
  - TO-broadcast(*m*);
  - TO-deliver(*m*), m é a mensagem.
- Quando um processo *p* efectua um *TO-broadcast*(*m*), dizemos que *p* TO-broadcasts *m*.
- Assumimos que cada mensagem *m* pode ser identificada pelo id (único) do emissor, denotado por *sender*(*m*).
- Adicionalmente assume-se que para qualquer mensagem *m*, e qualquer execução, *TO-broadcast*(*m*) só é executado no máximo 1 vez.

Atomic Broadcast:(cont.)

- Propriedades do TO-broadcast:
  - Validade (Para um processo *p* correcto: Se p TO-broadcasts m => p TO-delivers m);
  - Acordo Uniforme (Se p TO-delivers m->todos os p's correctos TO-deliver m);
  - Integridade Uniforme (Qualquer p TO-delivers m no máximo 1 vez e apenas se houve um TO-broadcast m pelo emissor);
  - Ordenação Total Uniforme (Se ambos os processos *p* e *q* TO-deliver das mensagens *m* e *m'*, então *p* TO-delivers *m* antes de *m'* sse *q* TO-delivers *m* antes de *m'*)

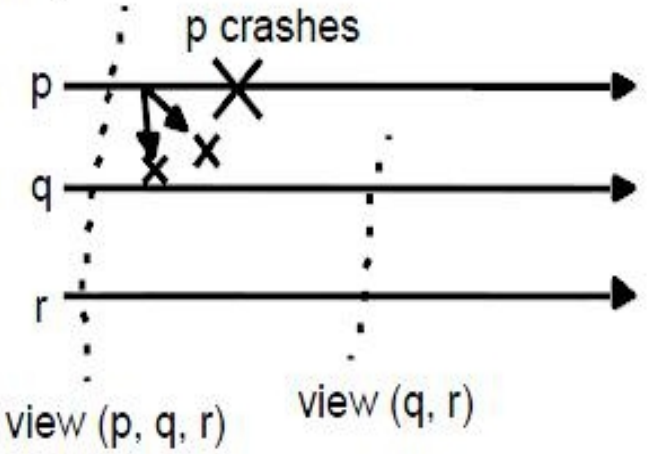


View Synchronous Broadcast: [4]

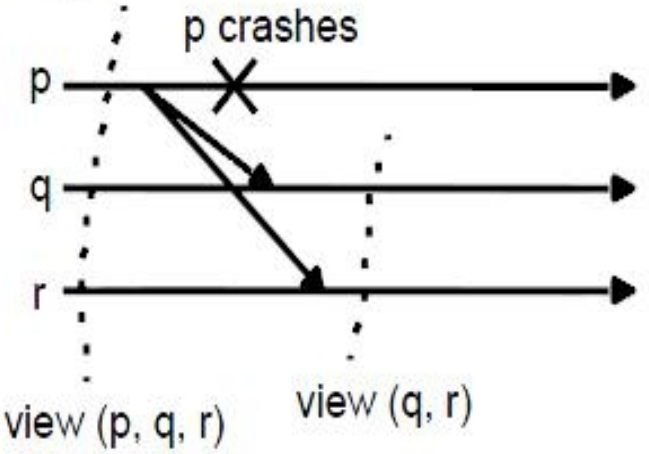
- O VS-CAST garante:
- Acordo: processos correctos entregam o mesmo conjunto de mensagens em qualquer view;
- Integridade: se um processo *p* entrega uma mensagem *m*, então esta não será entregue novamente pelo mesmo processo;
- Validade: processos correctos entregam sempre as mensagens que enviam.

View Synchronous Broadcast: (cont.)

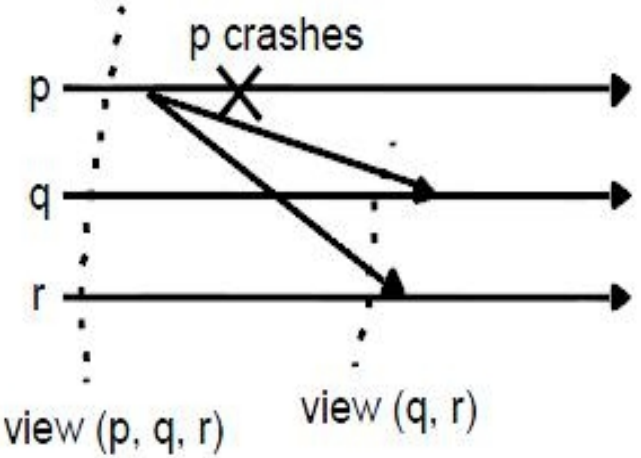
a (allowed).



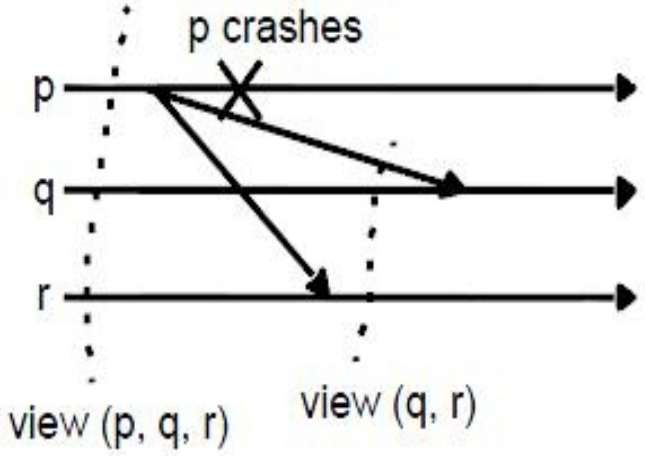
b (allowed).



c (disallowed).



d (disallowed).

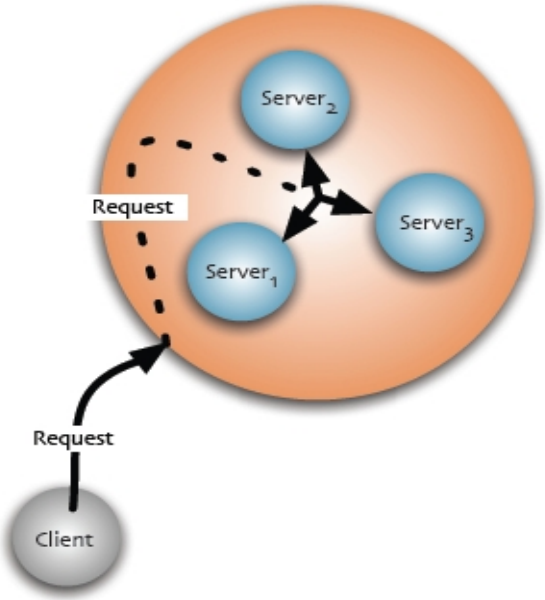


Métodos de replicação em S.D.

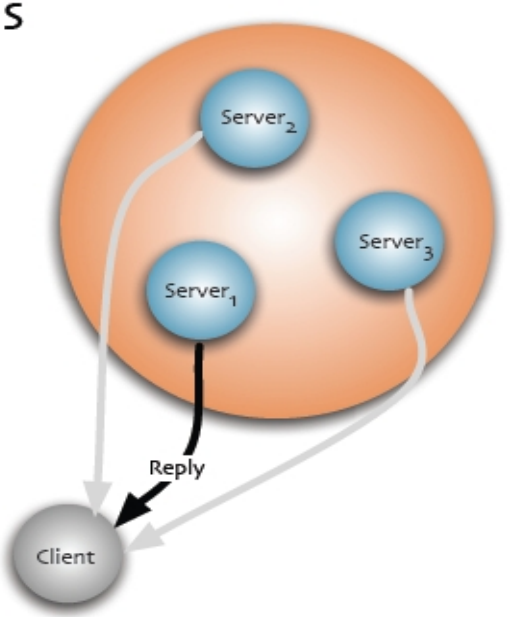
Replicação Activa: (state machine approach)

- Todas as réplicas recebem e processam a mesma sequência de pedidos dos clientes;
- Clientes não comunicam com um servidor particular mas vários como um grupo;
- Consistência é garantida assumindo que ao fornecer o mesmo input, e na mesma ordem, as réplicas irão produzir o mesmo output;
- Por forma a assegurar que os servidores recebm o mesmo input, e na mesma ordem, os clientes usam **ABCAST**.
- **Vantagens:** Simplicidade e Transparência nas falhas → As falhas são totalmente escondidas dos processos clientes, pois na eventualidade duma réplica falhar, os pedidos continuam a ser processados pelas restantes réplicas envolvidas no sistema.
- **Desvantagens:** Consomem muitos recursos pois temos a mesma carga de processamento em todas as réplicas.

Replicação Activa: (state machine approach)



2. Each replica processes the request, updates its own state, and returns the response to the client



1. Request goes to all the replicas

3. Client waits until it receives the first response (or a majority of identical responses)

Métodos de replicação em S.D. (cont.)

Replicação Passiva: (primary backup)

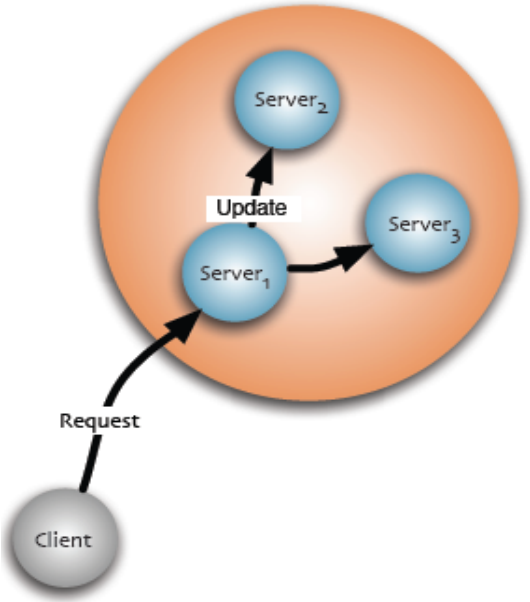
- Os clientes enviam os seus pedidos a um servidor/réplica primário, que os executa e transmite as alterações aos restantes (backups).
- Estes backups não executam os pedidos/invocações dos clientes, aplicando somente as alterações, resultantes da sua execução no servidor primário.
- A comunicação entre o servidor primário e os backups deve garantir que as actualizações são recebidas e processadas na mesma ordem (caso de um comunicação baseada em FIFO's).
- No entanto na eventualidade do servidor primário “crashar”, antes de todos os backups terem recebido as actualizações, para um dado pedido, o novo primário é designado sem no entanto, ter feito as actualizações.

Métodos de replicação em S.D. (cont.)

Replicação Passiva(cont.):

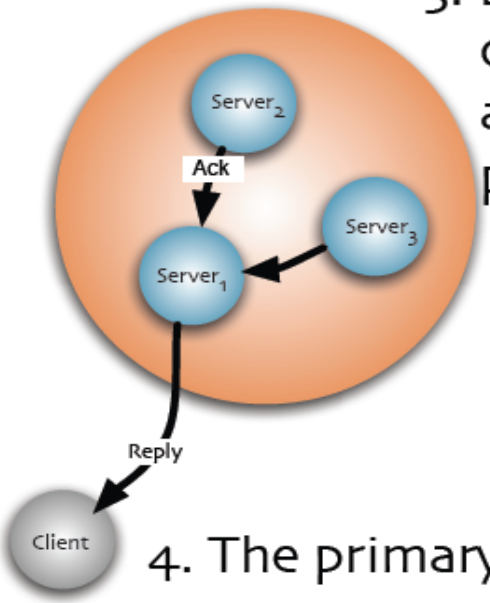
- É necessário um mecanismo que garanta que o novo primário envie as actualizações "devidamente" ordenadas, de acordo com as enviadas pelo primário que falhou. Um desses mecanismos é o **VSCAST** que é normalmente usado nesta técnica de replicação.
- **Vantagens:**
  - Consegue tolerar servidores não-determinísticos (ex: servidores multi-thread);
  - Usa muito menos poder de processamento, quando comparada com outras técnicas de replicação.
- **Desvantagens:**
  - Elevados custos de reconfiguração quando o servidor primário falha.

Replicação passiva: (primary-backup)



2. The primary processes the request, updates its own state, and sends a state update message to all other replicas.

3. Each replica updates its own state, and sends an acknowledgment to the primary.



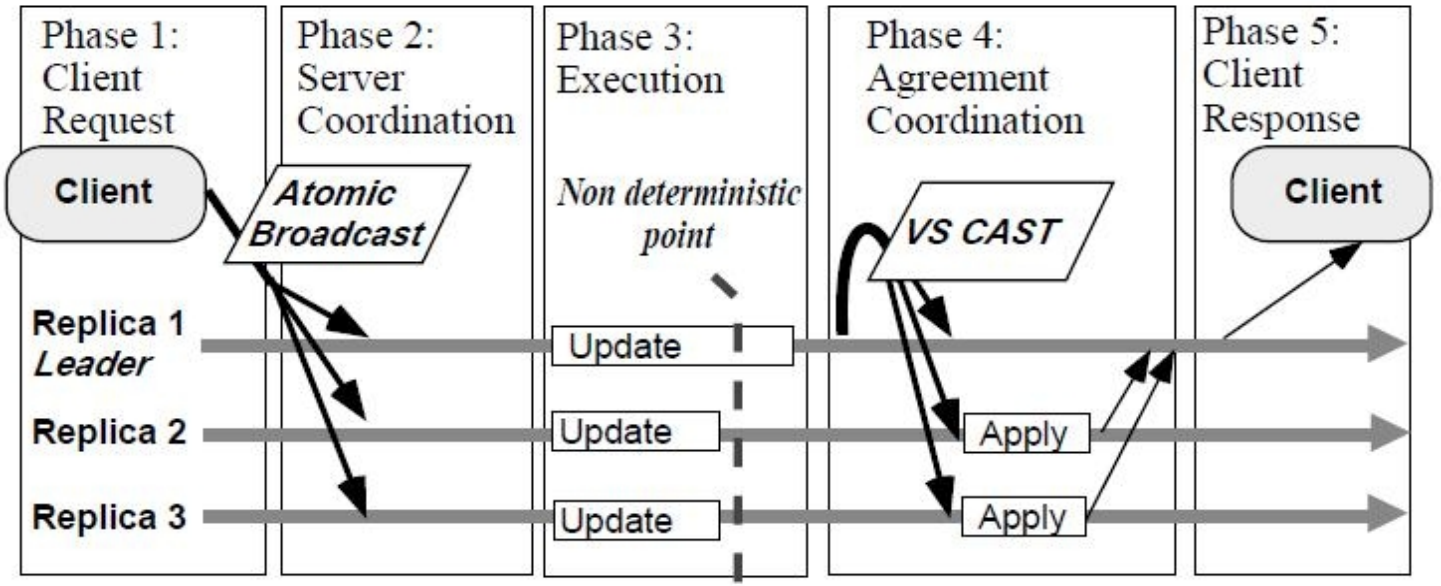
1. Request goes to a distinguished replica - the primary

4. The primary returns the response to the client.

Métodos de replicação em S.D. (cont.)

Replicação Semi-Activa:

- Solução intermédia entre a replicação activa e a passiva;
- A diferença em relação à replicação activa é que de cada vez que as réplicas têm que fazer uma decisão não-deterministica, uma delas (Leader) efectua a escolha e envia às restantes réplicas (Followers);
- As fases **EX** e **AC** são repetidas por cada escolha não-determinística.



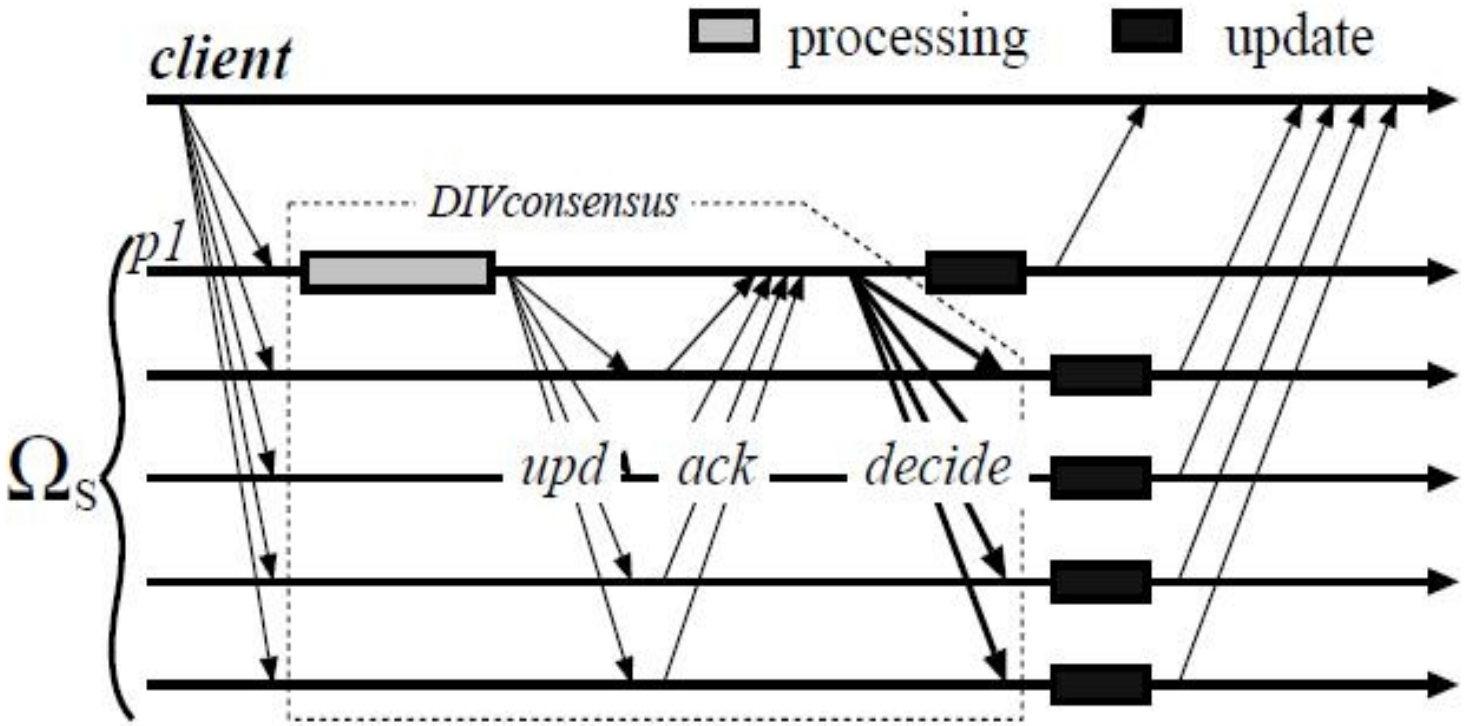
Replicação semi-activa. [1]

Métodos de replicação em S.D. (cont.)

Replicação Semi-Passiva:

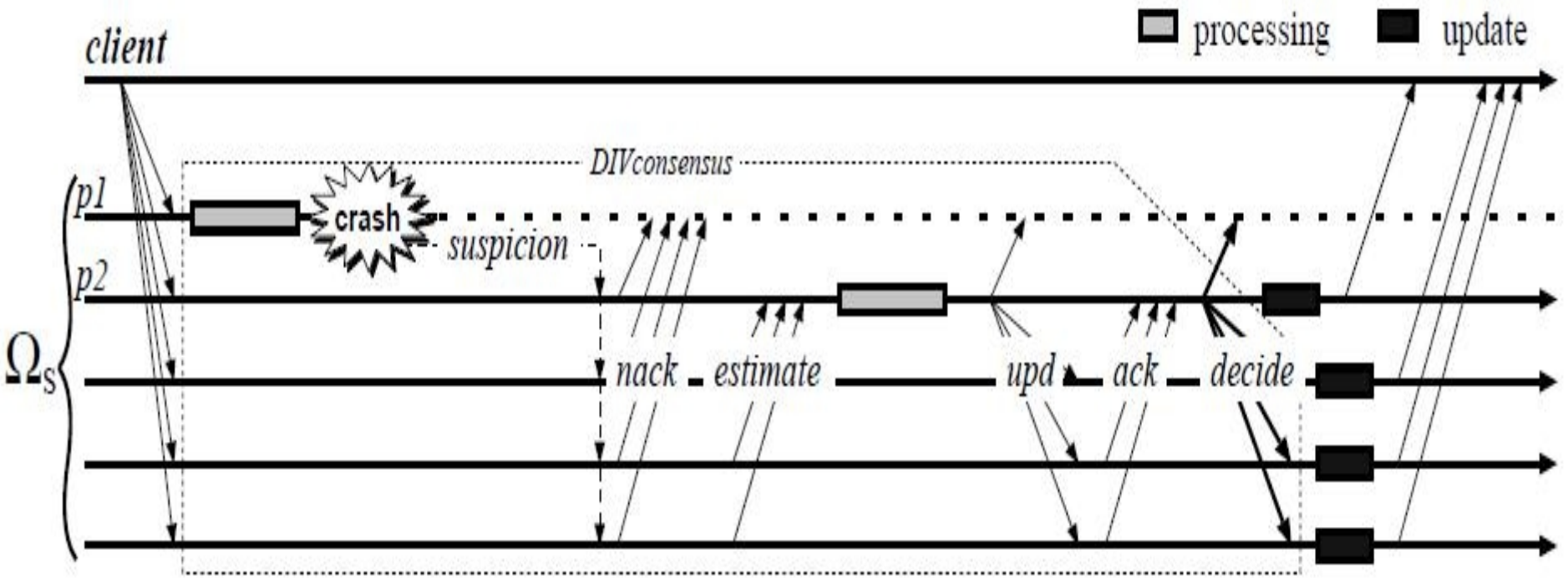
- Variante da replicação passiva;
- Como tal, nesta técnica existe um único servidor primário, que processa um ou vários pedidos do cliente, e seguidamente os backups são actualizados. No final todas as réplicas (primário e backups) enviam um resposta ao cliente.
- Um vez que tal acontece, o cliente não necessita de conhecer a identidade do servidor primário ou de possuir time-outs para detectar falhas neste.
- Tal como na replicação activa, nesta técnica o efeito de falhas no sistema é totalmente encoberto do cliente.
- A selecção do servidor primário, que processa os pedidos dos clientes, é baseada num paradigma de rotação do coordenador.
  - Este paradigma pode ser descrito como uma variante do problema do consenso entre servidores.
- 2 caracteristicas são mantidas em relação à replicação passiva:
  - Na ausência de falhas ou suspeitas de falha do servidor, o pedido é processado por apenas um servidor;
  - O processamento do pedido não necessita de ser deteminístico.

Replicação Semi-Passiva:



Replicação semi-passiva (boa execução)[2]

Replicação Semi-Passiva:



Replicação semi-passiva com 1 falha (pior caso)[2]

Replicação em Bases de Dados (BD)

Parte 1 - Aplicação do modelo de 5 fases a uma operação/transacção

Objectivos da replicação de BD

- Melhorar o desempenho e disponibilidade
- Diminuir tempos de resposta (eliminar o sobrecarga dos sítios remotos) em pedidos de leitura
- Melhorar a tolerância a falhas

Aplicação do modelo às BD

- Assumimos, nesta fase, que:
  - Uma transacção é uma operação simples, por exemplo, uma *stored procedure, select* (leitura)*, update* (escrita)
  - Uma transacção é atómica
  - A replicação da BD é para todos os dados e não só para algumas partes
  - Se um servidor falhar, as transacções serão efectuadas/abortadas nesse servidor mas o cliente poderá ligar-se a outro e refazer a transacção

Requisitos

- A transacção será efectuada (*commited*) ou abortada (*abort*) em todas as réplicas
- Se houver transacções concorrentes então devem de ser isoladas entre elas no caso de conflito (pe, escrita e leitura)
- O isolamento é feito através de protocolos de bloqueio – Two Phase Commit (2PC) - que garantam a serialização/linearização
- O cliente comunica exclusivamente com uma das réplicas

Estratégias de replicação de BD

Tipo de propagação da actualização
*Impaciente/Sincronizada vs Preguiçosa/*Assíncrona

Local da actualização
Primary copy vs Everywhere

Impaciente/Sincronizada (*Eager/Synchronous*) Cópia principal (*Primary copy*) 1	Preguiçosa/Assíncrona (*Lazy/Asynchronous*) Cópia principal (*Primary copy*) 3
Impaciente Actualiza em todo o lado (*Update Everywhere*) 2A 2B	Preguiçosa Actualiza em todo o lado (*Update Everywhere*) 4

Impaciente vs Preguiçosa (*Eager* vs *Lazy*)

- *Eager*: A actualização só será concluída depois de um número suficiente (quorum) de repostas de (*commit*)
- Vantagens: consistência dos dados; as alterações são atómicas
- Desvantagens: demasiadas mensagens, sobrecarga e fracos tempos nas actualizações

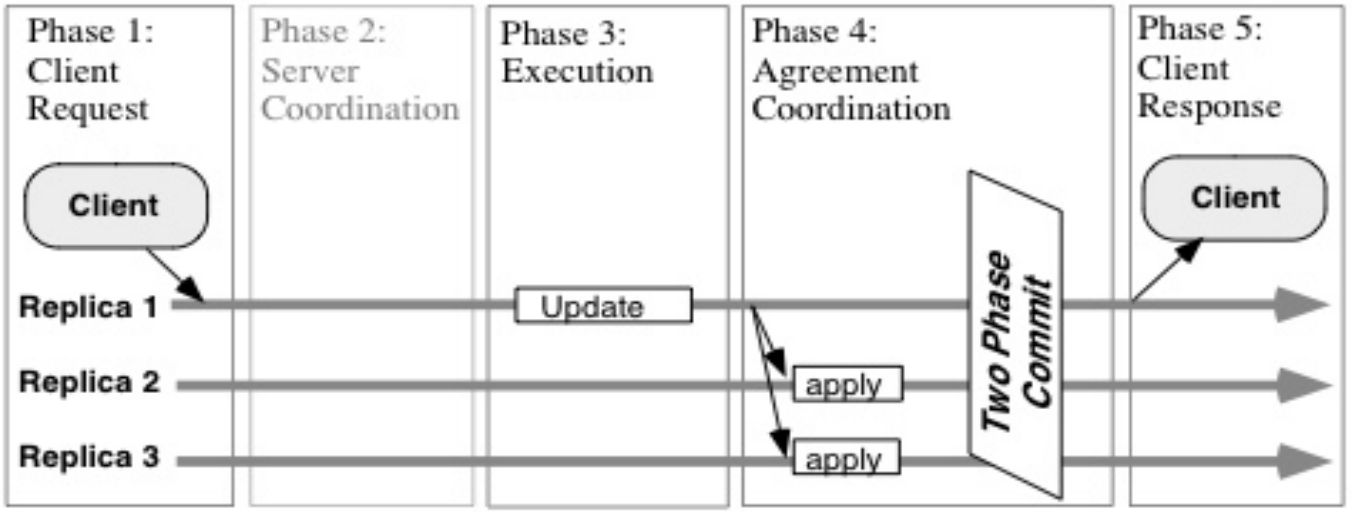
- *Lazy*: É actualizada a cópia principal, devolvida a resposta ao cliente, e só depois é feita a propagação pelas réplicas
- Vantagens: existem mais métodos para a propagação
- Desvantagens: Enquanto há propagação dos dados podem ocorrer inconsistências

Cópia principal vs Actualiza em todo o lado

- Actualização da cópia principal (*master*) e só depois as réplicas
- Vantagens: Existe controlo das réplicas, pelo menos o *master* tem as actualizações
- Desvantagens: a cópia principal pode ter sobrecarga, falhar, e, há um efeito de gargalo (bottleneck)

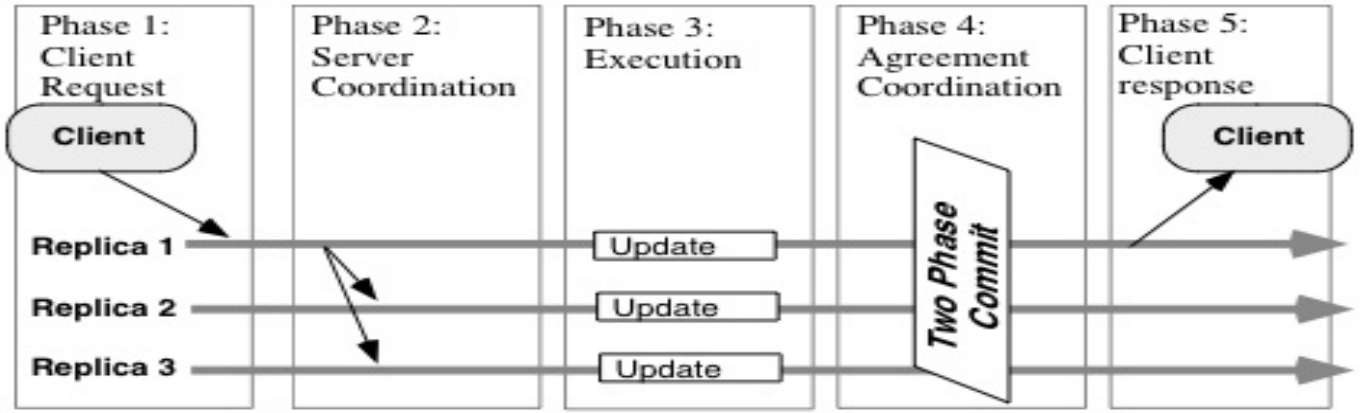
- As alterações podem ser inicializadas em qualquer uma das réplicas
- Vantagens: Rapidez de acesso e resposta, carga distribuída, qualquer réplica pode correr a transacção
- Desvantagens: Complexidade dos protocolos de actualização; réplicas precisam de ser actualizadas

1 – Impaciente na Cópia principal



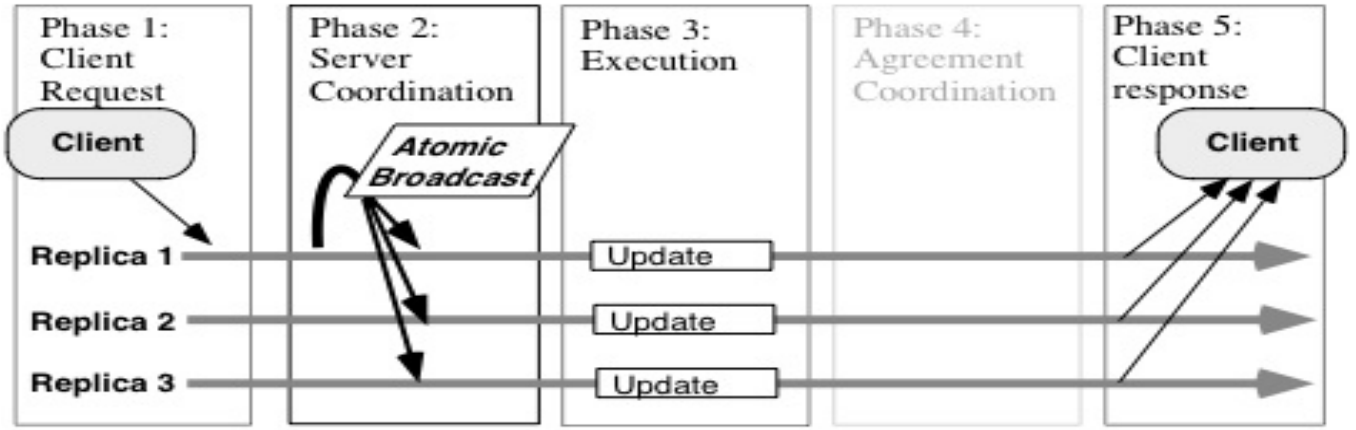
- Supressão da fase 2, 1º *update* na fase 3 e propagação na fase 4
- A resolução de conflitos é feita pela cópia principal que também efectua todas as operações
- Protocolo de bloqueio 2PC (1º *commit* para o acordo; 2º *commit* para a actualização), e, se um falhar aborta-se tudo
- As operações de leitura podem ser feitas em qualquer réplica porque dispõem da última versão
- É funcionalmente semelhante à replicação passiva dos SD onde o VSCAST é trocado pelo 2PC

2A - Impaciente, Actualiza em todo o lado com bloqueio distribuído (*distributed locking*)



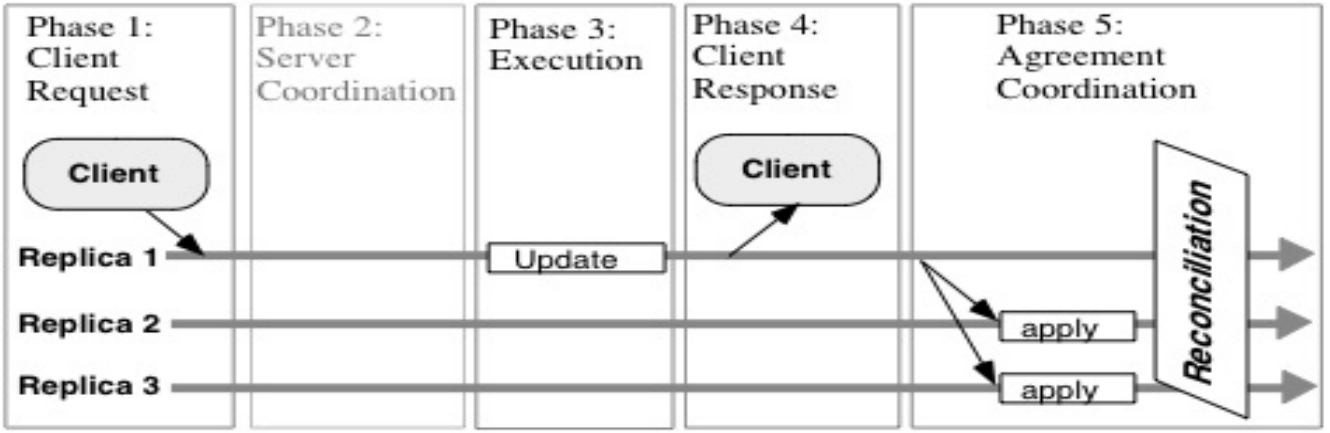
- Envio das mensagens na fase 2, a fase 3 é para a actualização, fase 4 reservada ao protocolo de bloqueio
- Acesso ao item só depois do bloqueio em todas as réplicas
- Método parecido à replicação semi-activa dos SD, excepto na fase 4 - Coordenação de Acordo – troca VSCAST por 2PC
- Nota: Se as BD fossem determinísticas (ie, mantém a ordem e sequência das operações) o 2PC não seria necessário

2B - Impaciente, Actualiza em todo o lado com *Atomic Broadcast - ABCAST*



- As diferenças para o 2A estão na: supressão da fase 4 e respectivo protocolo de bloqueio; na resposta do sistema ao cliente
- As transacções são efectuadas imediatamente em todas as réplicas com a garantia do ABCAST
- Se duas operações estiverem em conflito serão efectuadas pela ordem do ABCAST em todas as réplicas.
- Nota: Tentativa de uso das primitivas de comunicação em grupo

3 e 4 – Preguiçosa (*Lazy*) na Cópia Principal e Actualiza em todo o lado



- É mais rápido a responder já que troca as fases 4 e 5
- Evita a sobrecarga da replicação impaciente (*Eager*) e tem menos comunicações
- A diferença entre 3 e 4 é que, em 4 – *Update everywhere* -, qualquer réplica pode ser contactada pelo cliente
- Pode haver problemas de inconsistência entre as réplicas se a reconciliação de dados ainda não tiver ocorrido

Replicação em Bases de Dados (BD)

Parte 2 - Aplicação do modelo de 5 fases a transacções contendo várias operações

Transacções

- Assumimos, nesta fase, que: Uma transacção é composta por mais do que uma operação de leitura e escrita, e, que as operações podem não estar disponíveis em determinado tempo
- Ou seja, há casos novos, pelo que, implica protocolos novos a estudar (e que não existem para SD)
- Ao modelo anteriormente apresentado ocorrem mudanças consoante o protocolo, nomeadamente, na existência de um ciclo que juntará duas das fases:
  - Coordenação do Servidor e Execução
  - Execução e Coordenação do Acordo

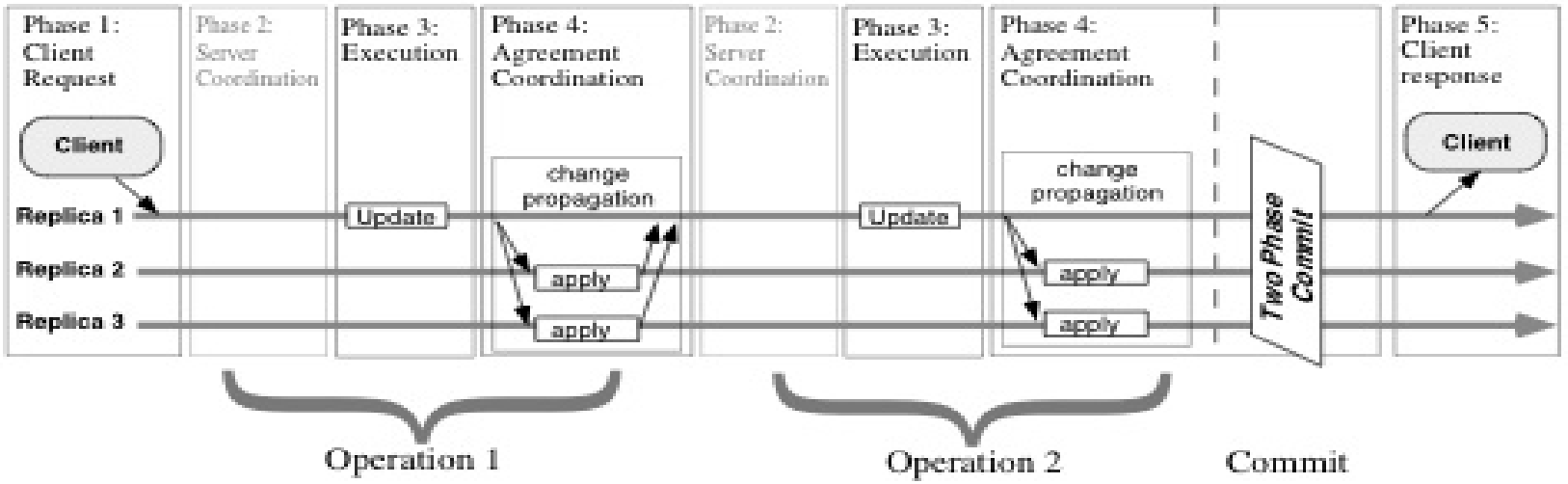
Estratégias para transacções

Tipo de propagação da actualização
*Impaciente/Sincronizada vs Preguiçosa/*Assíncrona

Local da actualização
Primary copy vs Everywhere

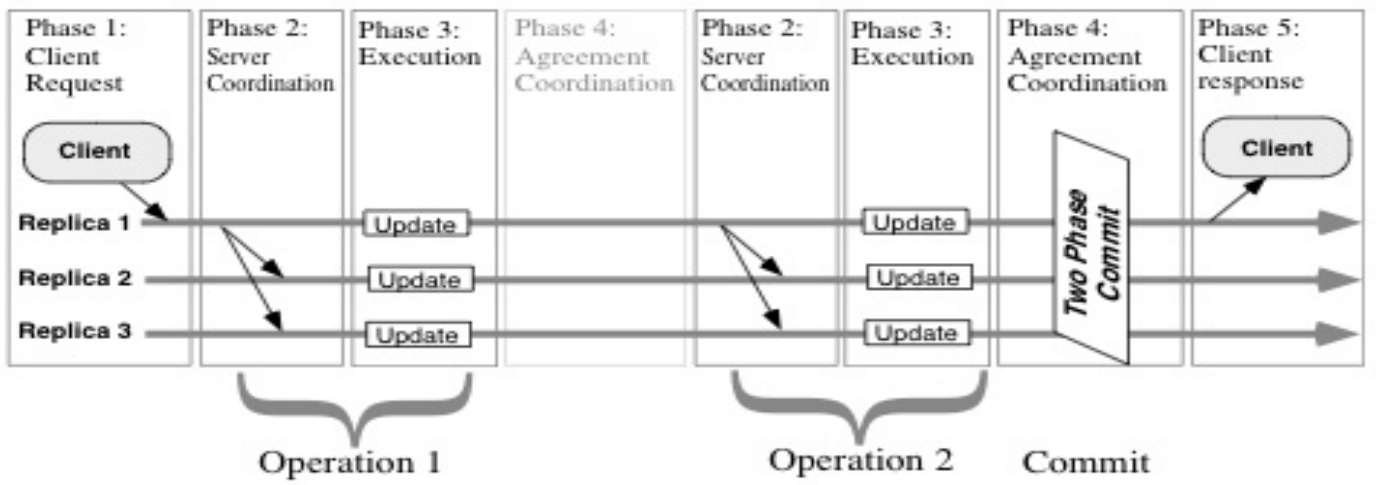
Impaciente/Sincronizada (*Eager/Synchronous*) 1 Cópia principal (*Primary copy*)	Preguiçosa/Assíncrona (*Lazy/Asynchronous*) 3 Cópia principal (Primary copy)
Impaciente Actualiza em todo o lado (*Update Everywhere*) 2A 2B	Preguiçosa Actualiza em todo o lado (*Update Everywhere*) 4

1 – Impaciente na Cópia principal



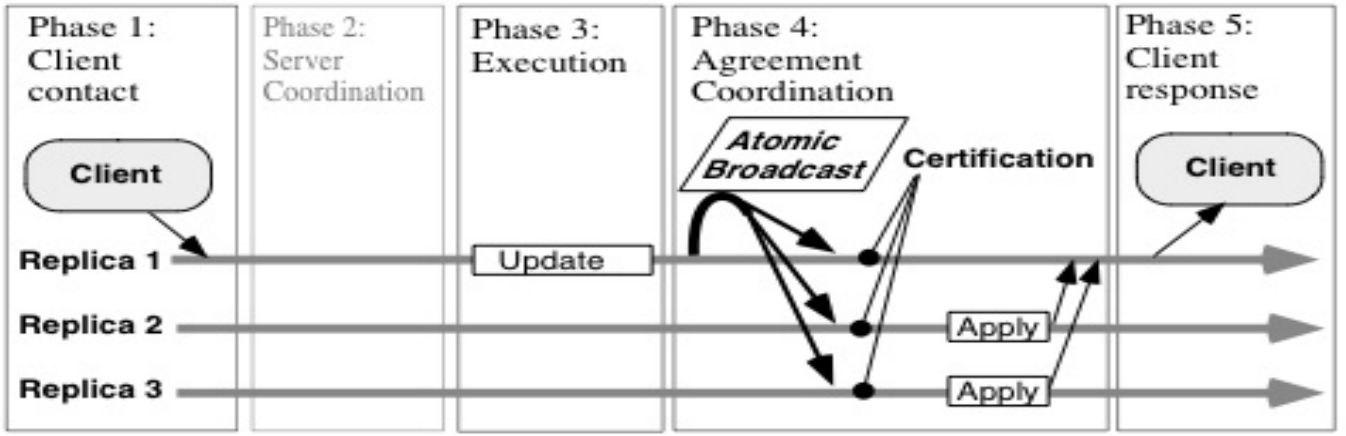
- Não há necessidade da fase 2: Coordenação do Servidor
- Os ciclos juntam as fases 3 e 4: Execução e Coordenação do Acordo
- As operações são feitas na réplica principal e posteriormente propagadas, e, no final, o protocolo de bloqueio 2PC garante sucesso e linearização
- Alternativa: Propagar as alterações da transacção depois desta estar completa (completa != *commited*), ie, ciclo de operações só durante a fase 3

2A - Impaciente, Actualiza em todo o lado com bloqueio distribuído (*distributed locking*)



- Os ciclos englobam as fases 2 e 3: Coordenação do Servidor e Execução
- Há um bloqueio para cada operação da transacção na fase 3 (não mostrado na figura)
- No final da transacção há um protocolo de bloqueio 2PC para garantir que tudo foi feito ou abortado

2B – Impaciente com Certificação



- Supressão da fase 2: Coordenação do servidor, e, uso do ABCAST na fase 4: Coordenação do acordo
- Uso do ABCAST para envio das operações para as réplicas
- A Certificação assegura que as transacções são executadas na ordem indicada pelo ABCAST
- A serialização não faz sentido pois as mensagens são enviadas separadamente
- A decisão de actualização é feita em conjunto

3 e 4 – Replicação preguiçosa para Transacções

- As actualizações não são propagadas (completas) sem que as transacções estejam efectuadas (*commit*)
- Não importa se a transacção tem mais do que uma operação
- As actualizações são feitas como um todo porque são feitas *a posteriori*

Consistência dos métodos (BD) para Parte 1 e 2

Forte

1 - Impaciente na Cópia principal

2 - Impaciente e Actualiza em todo o lado:

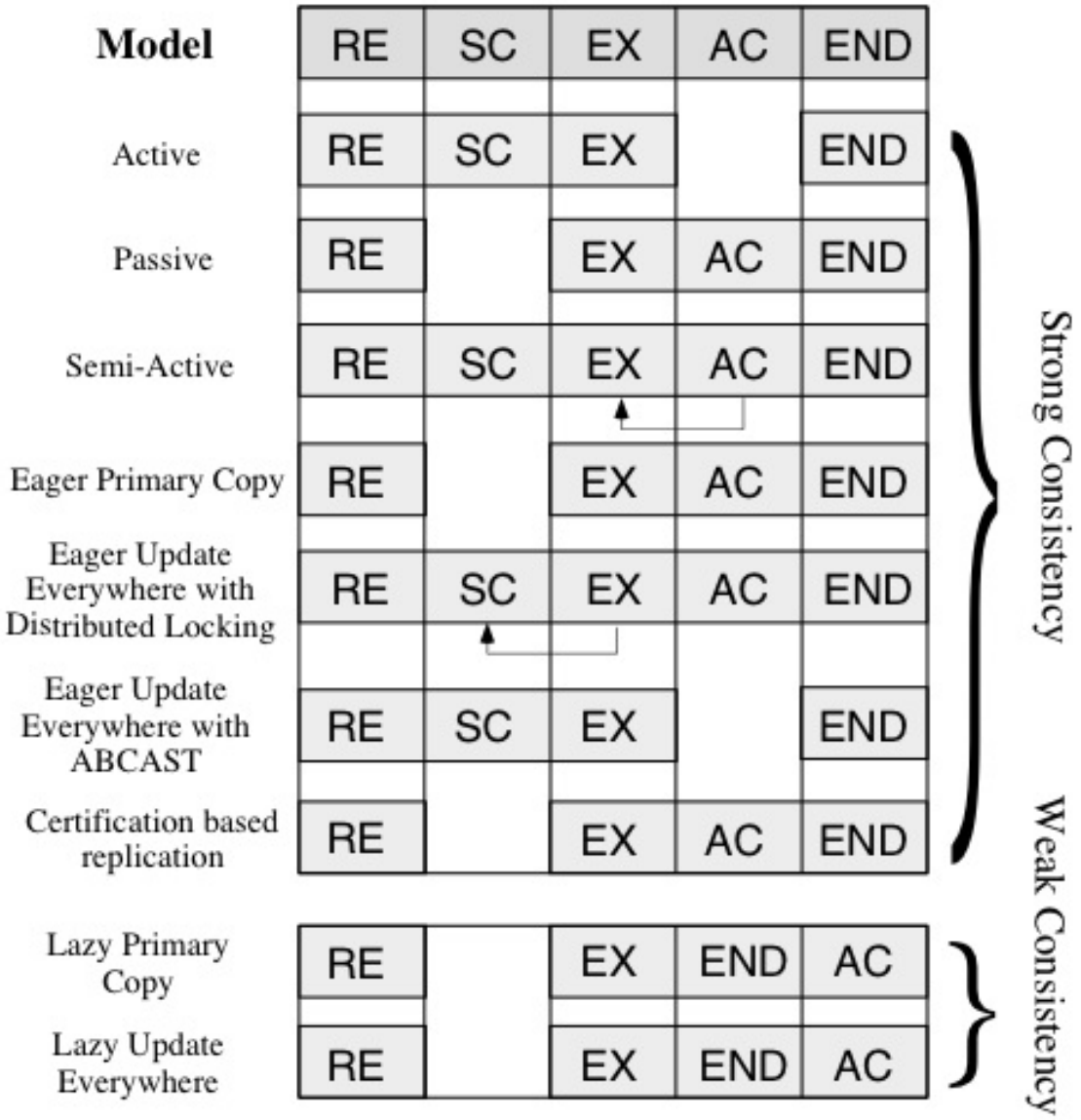
- Com bloqueio distribuído
- Atomic Broadcast
- Impaciente com certificação

Fraca

3 - Preguiçosa na Cópia principal

4 - Preguiçosa e Actualiza em todo o lado

Métodos (SD e BD) e Consistência



Programação Paralela com Memória Transaccional

Programação Paralela com Memória Transaccional
- Resumo -

- Introdução

Porquê paralelizar.

- Problemas de Sincronização

Divisão gera problemas adicionais.

- Dilema do Programador

Paralelo *versus* a complexidade. Consistência.

- Procurando noutro lugar

Problemas de consistência não são novos.

- Memória Transaccional

Transições aplicadas a operações de memória.

- O Problema

Múltiplas *threads* executam várias funções

- Validação e Fidelidade

Determinar se uma *memory location* é usada como parte de outra implementação

- Memória Transaccional na Actualidade

As primeiras implementações.

Programação Paralela com Memória Transaccional
- Introdução -

- A velocidade dos processadores não tem aumentado ao nível verificado nas últimas décadas.

  A necesidade de maior velocidade é real.

- As cada vez mais complexas aplicações levam os programadores a procurar alternativas.

  Surgem os processadores de vários *cores*.

- É necessário paralelizar!

  Múltiplos caminhos de execução terão de cooperar para completar tarefas do programa, e parte acontecerá de forma concorrente.

- Lei de Amdahl.

  Modelo de relação entre a velocidade esperada de um algoritmo implementado de forma paralela relativa ao algoritmo normal.

Programação Paralela com Memória Transaccional
- Problemas de Sincronização -

- Dividir um programa para que possa ser executado gera um conjuntos de problemas adicionais:
  - Diversas partes do programa tem de colaborar. Resulta na partilha de memória.
  - Acesso a memória partilhada tem de ocorrer de forma controlada.
    - Um programa não pode atingir um estado inconsistente.
    - Se um estado é representado pelo conteúdo de múltiplas localizações de memória surgem problemas.
    - Um processador não consegue modificar atomicamente uma localização.

Programação Paralela com Memória Transaccional
- Problemas de Sincronização -

- Dividir um programa para que possa ser executado gera um conjuntos de problemas adicionais:
  - Programação paralela recorre à sincronização para trabalhar com múltiplas localizações de memória.
    - *Mutex*: garante ao programa que este está a executar sozinho uma operação protegida, garantindo não verá um estado inconsistente.
  - Programas com *mutex* simples veêm a sua performance diminuída.
    - Parte do programa que é executa em paralelo é menor.
  - O uso de mais *mutexes* resolve essa questão assim como o sobrecarregamento de *locking* e *unlocking mutexes*.
    - Podem surgir *deadlocks*.

Programação Paralela com Memória Transaccional
- Dilema do Programador -

- O programador depara-se entre duas situações:
  - Aumentar a parte do programa em paralelo
  - Ou aumentar a complexidade do código (e mais problemas).

- Paralelização sinónimo de Problemas? Não, obrigado.
  - Experiência do programador é determinante.

- Paralelizar um projecto inteiro de uma só vez é irreal!
  - Processo longo que requer testes constantes.
  - *Locks* maiores são de resolução mais difícil.

- Ilação: *locking* é uma aproximação errada para resolver os problemas de consistência.

Programação Paralela com Memória Transaccional
- Procurando noutro Lugar -

- Problema de consistência não é novo.
  - Problema central nas base de dados.
  - Uma base de dados tem ser actualizada atomicamente.
  - As transações apresentam-se como a solução no mundo das BD.

- Programadores explicitam quais as operações que pertencem a uma transição.
  - Operações na transação ocorrem de forma arbitrária mas não tem qualquer efeito até a transição terminar.
  - Se surgirem conflitos, a transição retorna ao estado anterior e é reiniciada.

- Se as alterações de uma transição são feitas atomicamente em simultâneo, então a ordem em que as alterações são adicionadas à transição é irrelevante.

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Uma transação é uma sequência de operações de leitura e escrita em memória partilhada que impõe modificações no sistema. [6]
  - Assume que os estados intermediários não foram alterados concorrentemente por outras tarefas.

- Características:
  - Atomicidade. *All-or-nothing*.
  - Consistência. Manter a integridade.
  - Isolamento. Ilusão de que não existem outras transições concorrentes; sistema impede uma transação de actuar sobre um estado inconsistente gerado por outra transação.
  - Duradoura.

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- A memória Transaccional (TM) reduz a complexidade da programação paralela abstraíndo a necessidade de mecanismos de sincronização do código em paralelo [7].
  - Programadores podem escrever código sem se preocupar com *deadlocks*, *livelocks* ou *race conditions* [8].
  - Abdicando do uso de mecanismos é possível programar por módulos. [9]

- O conceito pode ser comparado às transações de base de dados.
  - Controla o acesso à memória partilhada em programação concorrente.
  - Simplifica a programação paralela permitindo que um grupo carrege e aloje instruções para as executar de forma atómica.

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Remonta a 1993: Memória Transaccional de Herlihy e Moss
  - Implementação *hardware* (HTM).

- Actualmente, há falta de suporte para HTM
  - Software Transactional Memory (STM) é mais usado pelos projectos de pesquisa actuais.
    - Permitem interfaces funcionais com a TM, o que faz com que os programadores consigam usar as simplificações TM mesmo sem o suporte HTM.

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Vamos supôr que um código de programa é *thread safe.*
  - múltiplas *threads* executam de forma concorrente, sem produzir um resultado inválido.
- Os resultados são os esperados mas o desempenho está longe de ser óptimo.
  - **f1_1** e **f2_2** são as funções mais usadas e podem coexistir sem qualquer problema.
  - um *lock* desnecessário atrasa o programa.

```
long counter1; long counter2;
time_t timestamp1; time_t timestamp2;

void f1_1(long *r, time_t *t)){
*t = timestamp1; *r = counter1++; }
void f2_2(long *r, time_t *t)){
*t = timestamp2; *r = counter2++; }
void w1_2(long *r, time_t *t)){
*r = counter1++;
if(*r & 1)      *t = timestamp2;}
void w2_1(long *r, time_t *t)){
*r = counter2++;
if(*r & 1)      *t = timestamp1;}
```

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Primeira alternativa:
  - 4 *locks*, um para cada par de variáveis.
  - **w1_2** produz resultados inconsistentes (atrasa a obtenção de **l_timestamp1**).
- Não serve!

```
void f1_1 (long *r, time_t *t){
lock(l_timestap1);
lock(l_counter1);
*t = timestamp1;
*r = counter1++;}

void w1_2 (long *r, time_t *t){
lock (l_counter1);
*r = counter1++;
if (*r & 1){
lock(l_timestamp1);
*t = timestamp2;
unlock (l_timestamp1);}
      unlock(l_counter1);}
```

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Tentativa de obter os dois locks em **w1_2** numa ordem diferente de **f1_1**
  - Leva a *deadlocks*!
- É fácil chegar a uma situação onde diversos *mutex locks* são necessários para a paralelização.
  - E usá-los correctamente é complicado.
- TM apresenta solução.

```
void w1_2 (long *r, time_t *t){
lock (l_counter1);
lock (l_timestamp1);
*r = counter1++;
if (*r & 1){
*t = timestamp2;
unlock (l_timestamp1);
                    unlock(l_counter1);}
}
```

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- **tm_atomic**: todas as instruções do bloco fazem parte de uma transação.

- Chamar funções pode ser um desafio, uma vez que as funções também devem ser *transaction aware*.
  - *Nesting:* chamar um bloco atómico dentro de outro.
  - *Nesting* tem de ser gerido.

```
void f1_1(long *r, time_t *t);
void f2_2(long *r, time_t *t){
tm_atomic{
*t = timestamp1;
*r = counter1++;}
}
void w1_2(long *r, time_t *t);
void w1_2(long *r, time_t *t){
tm_atomic{
*r = counter2++;
if(*r & 1)
*t = timestamp1;}
}
```

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- *Nesting* [7]:
  - *Flat nesting*
    - Transacção interior merge por completo com a exterior.
    - Simplicidade.
    - Se o bloco interior falha, toda a transação é anulada.
  - *Closed nesting*
    - Início de cada bloco atómico funciona como *checkpoint*; em caso de conflito com outra transação, é feito o *rollback* e reiniciada.
    - Mais produtiva e de fácil implementação.
    - Mais complexa.
  - *Open nesting*
    - Transações feitas no *top level* (visíveis ao mundo).
    - Permite um aumento brutal de paralelismo; no entanto, anular uma transação é complicado porque já foi commited.
    - Muito complexa, quer em termos de programação quer implementação **(recomendada a quem sabe mesmo o que faz).**

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Tratar do *nesting*:
  - 1. verificar se a localização da memória faz parte de uma outra
  - 2. caso faça, cancela-se a transação.
    - ou a transacção pode ser feita até ao fim e depois as alterações são desfeitas.
  - 3. caso não faça, registar a localização para que uma outra eventual transacção saiba disso.
    - falha se a transacção anterior acedeu à mesma localização.
  - 4. dois casos: leitura e escrita.
    - Leitura: se a variável não foi modificada, faz-se o *load* do valor da localização da memória; se a variável foi modificada, o *load* é feito do armazenamento local.
    - Escrita: escreve a variável no armazenamento local.

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Quando o bloco **tm_atomic** é acedido:
  - 1. Se a transacção foi abortada, o estado interno deverá sofrer um *reset*, atrasado por um período curto, e posteriormente, o bloco deve ser re-executado.
  - 2. Guardar todos os valores da localização de memória modificadas na transacção para os quais os novos valores foram guardados no armazenamento local.
  - 3. Libertar as localizações de memória: *reset* à informação guardada sobre as localizações de memória que fazem parte da transacção.
- Implementar tudo de forma eficiente é um desafio.

Programação Paralela com Memória Transaccional
- Memória Transaccional -

- Outro Exemplo [6].

```
bool mutexX,
mutexY;

exemploLock (x,
y)
{
    lock(mutexX);
    lock(mutexY);
    x = x + y;
    y = 0;

unlock(mutexX);

unlock(mutexY);
}
```

```
exemploAtomic (x,
y)
{
    atomic {
        x = x + y;
        y = 0;
    }
}
```

Programação Paralela com Memória Transaccional
- Validação e Fidelidade -

- Com uma implementação correcta podemos determinar se uma localização de memória está a ser usada por outra implementação.
  - E saber que não produzimos resultados inconsistentes.

- Transacção será executada se apenas a memória aceder com sucesso a um bloco **tm_atomic**, e a transacção não for anulada.

- Será que as *threads* ao usar a tecnologia TM terminarão, se estiverem constantemente abortando umas às outras?

Programação Paralela com Memória Transaccional
- Validação e Fidelidade -

- Exemplo para comparação:
  - Numa rede baseada em IP, todas as máquinas ligadas podem enviar ou receber em simultâneo em qualquer momento.
    - Quando detectado um conflito, o envio é reiniciado após um pequeno momento.
  - O protocolo IP tem ainda um algoritmo exponencial de *backup* que as *stacks* tem de implementar.
  - Vivemos num mundo baseado em redes IP, logo esta abordagem deverá ter sucesso.

Programação Paralela com Memória Transaccional
- Validação e Fidelidade -

- 1º códido rejeitado previamente!

- Com o uso de Memória Transaccional:

  - as funções usam memórias disjuntas.
  - as localizações de memórias acedidas também são distintas, e, como tal, a verificação pelo acesso concorrente nunca abortará o processo.

- Assim, é teoricamente possível resolver a questão com a TM.

Programação Paralela com Memória Transaccional
- TM na Actualidade -

- VELOX
  - Objectivo: compreensão analítica dos locais de um sistema operativo onde a TM pode ser usada (desde estruturas de dados livres a aplicações de alto nível).
  - TM tem suporte de *hardware*.
  - Pesquisa ainda as semânticas mais usadas para que possam ser adicionadas a linguagens de programação de alto nível.
- TinySTM
  - Oferece as primitivas necessárias para Memória Transaccional portátil, compacta e dependente de apenas alguns serviços.
  - Originará extensões de linguagem em alguns compiladores.

Programação Paralela com Memória Transaccional
- TM na Actualidade -

- As extensões adicionadas às linguagens de programação vão permitir recolher experiência com o uso da Memória Transaccional em situações reais.

- Existem tópicos em aberto que requerem mais pesquisa:

  - Registar as transacções.
  - Gestão das transacções anuladas.
  - Semântica.
  - Desempenho.

Programação Paralela com Memória Transaccional
- TM na Actualidade -

- Registar as transacções:
  - Guarda a localização exacta da memória reservada usa na transacção.
  - Ineficiente: armazenamento excessivo de várias palavras para cada espaço usado.
    - Se um espaço de memória está um bloco já registado, não há necessidade de o registar
    - Levanta questões debatidas anteriormente: **f1_1** e **f2_2** deixam de ser independentes se as variáveis estiverem num mesmo bloco.

Programação Paralela com Memória Transaccional
- TM na Actualidade -

- Gestão das transições anuladas:
  - *Lazy abort/ Lazy commit*: transacção continua a ser executada, sendo os resultados escritos na memória apenas depois da transacção completar com sucesso.
  - *Eager/Eager*: transacção é abortada o mais cedo possível, e reiniciada se necessário.
    - Valores são imediatamente guardados: o valor antigo na localização de memória é guardado na memória local, para que possa ser restaurado se necessário.
  - Há mais alternativas para tratar os detalhes.
    - E uma maneira pode não ser suficiente.

Programação Paralela com Memória Transaccional
- TM na Actualidade -

- Semântica:
  - Semântica do bloco **tm_atomic** tem de ser especificada e integrar a Memória Transaccional com o resto da linguagem.
  - *Nesting*.
  - Variáveis locais.
    - Podem não fazer parte da transação mas será necessário um *reset* às mesmas caso seja anulada.

Programação Paralela com Memória Transaccional
- TM na Actualidade -

- Desempenho:
  - Grande parte das optimizações podem e devem ser feitas pelos compiladores.
  - Problema: o código de um programa é usado em diversas situações e o processamento/armazenamento em excesso com a TM é elevado.
  - Torna-se necessário gerar duas versões do mesmo código: uma com suporte TM e outra sem.
    - A versão com suporte TM tem de garantir que a versão sem suporte seja utilizado o mais frequente.

Conclusões: Replicação (SD e BD)

- Apesar dos modelos, restrições e terminologias os algoritmos para a replicação em SD e BD são semelhantes;
- A replicação em SD tem como objectivo melhorar a tolerância a falhas; A replicação em BD teve, inicialmente, como objectivo melhorar o desempenho e disponibilidade das mesmas;
- Os sistemas de replicação de BD beneficiaram da abstracção e dos algoritmos de comunicação em grupo que existe para SD;
- Para replicação em BD os métodos *synchronous* (*Eager*) são mais complicados de implementar por causa das características de serialização (determinísticos), de consistência e pelos protocolos de bloqueio. Mas aumentam o tempo de espera (*time-delay*) e comunicações (*bottleneck*).

Conclusões: Programação Paralela com Memória Transaccional

- Memória Transiccional promete simplificar a programação paralela.
- O conceito de transacções aplicado ao mundo da Informática é usado em diversos programas e provou ser razoavelmente fácil de entender para os programadores.
- Existem algumas implementações, mas estão longe de estar finalizadas.

Bibliografia:

- [1] Understanding Replication in Databases and Distributed Systems, M. Wiesmann, F. Pedone, A. Schiper, B. Kemme andG. Alonso, Proc. of Int'l on Distributed Computing Systems (ICDCS 2000), pp. 264-274, 2
- [2] Défago, X., Schiper, A., and Sergent, N. 1998. Semi-Passive Replication. In Proceedings of the the 17th IEEE Symposium on Reliable Distributed Systems (October 20 - 23, 1998). SRDS. IEEE Computer Society, Washington, DC, 43.
- [3] Défago, X., Schiper, A., and Urbán, P. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36, 4 (Dec. 2004), 372-421.
- [4] Distributed Systems: Concepts and Design (3rd Edition), Jean Dollimore, Tim Kindberg , George Coulouris.
- [5] Wiesmann, M., Défago, X., and Schiper, A. 2003. Group Communication based on Standard Interfaces. In Proceedings of the Second IEEE international Symposium on Network Computing and Applications (April 16 - 18, 2003). NCA. IEEE Computer Society, Washington, DC, 140.

Bibliografia: (cont.)

- [6] Memórias Transacionais, Izumi Oniki Chiquito, Rafael Auler;
- [7] Nesting Models in Software Transactional Memory: http://everything2.com/title/Nesting%2520Models%2520in%2520Software%2520Transactional%2520Memory;
- [8] Configurable Transactional Memory, Christoforos Kachris, Computer Engineering Department, Delft University of Technology, The Netherlands, Chidamber Kulkarni, Xilinx Research Labs, Xilinx Inc., San Jose, CA;
- [9] Beautiful Concurrency, Simon Peyton Jones, Microsoft Research, Cambridge;
- [10] Matthias Wiesmann, Fernando Pedonet, Andr Schipper, Bettina Kemmet, Gustavo Alonso. Database replication techniques: a three parameter classification. IEEE Symposium SRDS2000. 2000;
- [11] J. Gray et al. The Dangers of Replication and a Solution: www.iiit.net/~pkreddy/adbms03/Lectures/Replication_adbms.ppt;