Replicação e Comunicação de Grupo

Objetivo

- Mostrar relação entre TF baseada em software (gerência de replicação baseada em software) e CG, indicando CG como um *framework* adequado

Modelo de falhas considerado

- SD: processos, canais e mensagens
- processo: correto/incorreto (especificações)

  incorreto falha por *crash* (*send/receive*)
- canais: assíncronos (*not bounded*) e confiáveis (se processos estão corretos, mensagem *eventualmente* chega ao destino).

  Defeitos no canal *eventualmente* são reparados.

Consistência da réplica

- Modelo: após fazer a invocação o processo é bloqueado até receber a resposta
- Critérios de consistência: def. estado mais recente
  - causal (*weak*), seqüencial e linearizabilidade (*strong*)
- Escolha: linearizabilidade ou *one-copy* (dá ao cliente a ilusão de servidores não replicados)
- Motivo: mais fácil de implementar do que a seqüencial
- Propriedades exigidas: Ordem e Atomicidade

Técnicas de replicação

Primary-backup

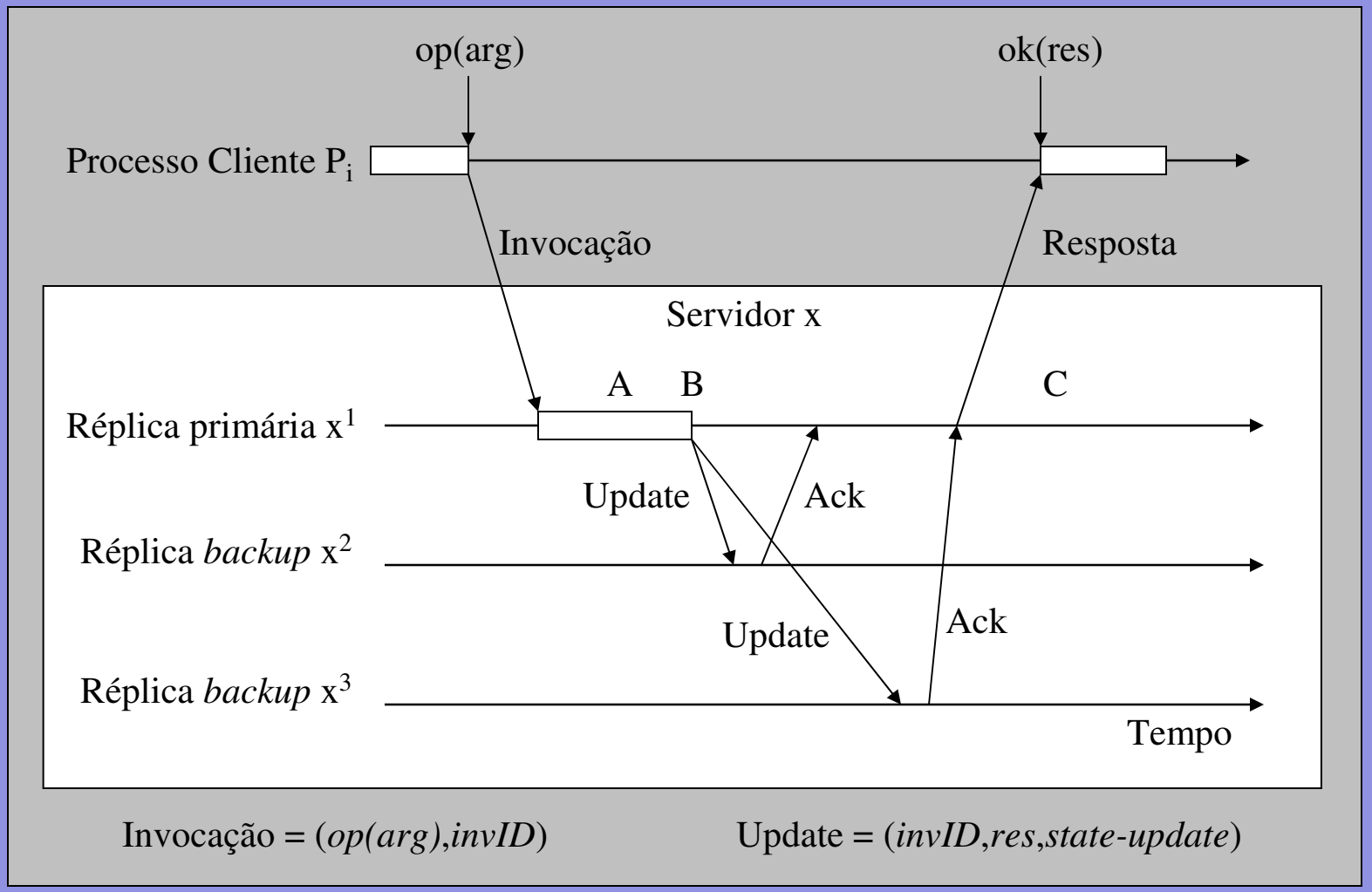
e

Ativa

Replicação *Primary-backup*

- Procedimento: primário recebe invocação, processa, atualiza estado dos *backups* e retorna resposta ao cliente
- Linearizabilidade garantida por:
  - ordem: estabelecida pelo primário
  - atomicidade: controlada pela recepção das mensagens de atualização de estado
- Pontos de falha: *backups* ou primário

Replicação *Primary-Backup*



Falhas do primário

- Necessita-se eleger um novo primário

- Casos de falhas:
  - A - antes de enviar *update*;
  - B - durante ou após *update*, mas antes da resposta; e
  - C - após cliente receber resposta.

Falha em A

- Cliente não recebe resposta e suspeita que servidor está falho;
- Cliente instrumenta-se de novo *invID* e refaz invocação;
- Novo primário considera invocação como nova e segue adiante.

Falha em B

- Cliente não recebe resposta, suspeita que servidor está falho, instrumenta-se de novo *invID* e refaz invocação;

- Necessita atomicidade na comunicação
  - nenhum recebe *update* - reporta ao caso A;
  - todos recebem *update* mas cliente não recebe resposta (gera necessidade de guardar a tupla (*invID*,*res*)).

    Ao receber novamente invocação primário retorna *res*.

Falha em C

- *Crash* é transparente para o usuário

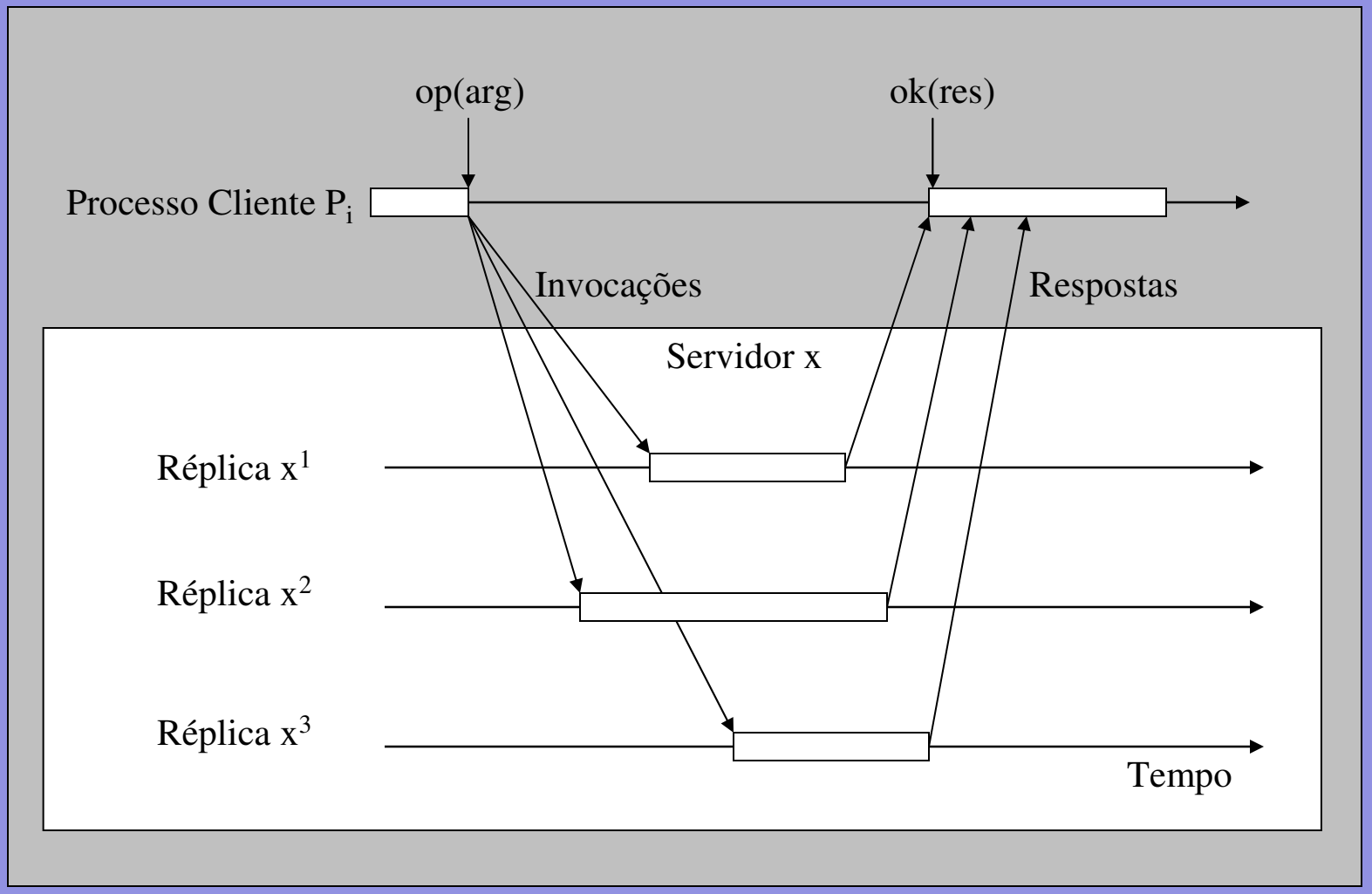
Considerações preliminares

- Técnica depende de mecanismo de detecção determinístico (consenso), impossível em sistemas puramente assíncronos
- correteza da técnica com detectores não confiáveis requer primitiva de *multicast* com visão síncrona (VSCAST)

Replicação Ativa

- Todas réplicas recebem invocação, processam, atualizam estado e retornam resposta ao cliente
- dependendo da semântica de falhas suportada o cliente espera por uma ou pela maioria das respostas
- Ponto de falha: réplicas (transparente)

Replicação Ativa



Considerações preliminares

- requer que todas as réplicas recebam as invocações na mesma ordem;
- *multicast* das invocações também deve ser atômico
- primitiva de *multicast* totalmente ordenado (TOCAST) fornece requisitos

Comparação entre as técnicas

- Replicação ativa depende do estado inicial e da seqüência de operações sobre as réplicas, enquanto que a *primary-backup* não.

- O *crash* de uma réplica (ativa) ou *backup* é transparente para o cliente enquanto que o *crash* da réplica primária não.

Comunicação de Grupo

- Abstração conveniente: um grupo pode representar o conjunto de réplicas do servidor *x*
- Vantagem: propriedades da comunicação (*multicast* atômico, ordenado e confiável) facilitam a implementação das técnicas de replicação

Grupos estáticos e dinâmicos

- Estáticos: nunca mudam seus membros. Membros falhos permanecem no grupo
- Dinâmicos: usa conceito de visão. Permite *join* e *leave* dinamicamente
- *Primary-backup* exige grupo dinâmico, enquanto replicação ativa não
- Maioria das implementações de replicação ativa usam grupos dinâmicos porque *multicast* totalmente ordenado requer isto

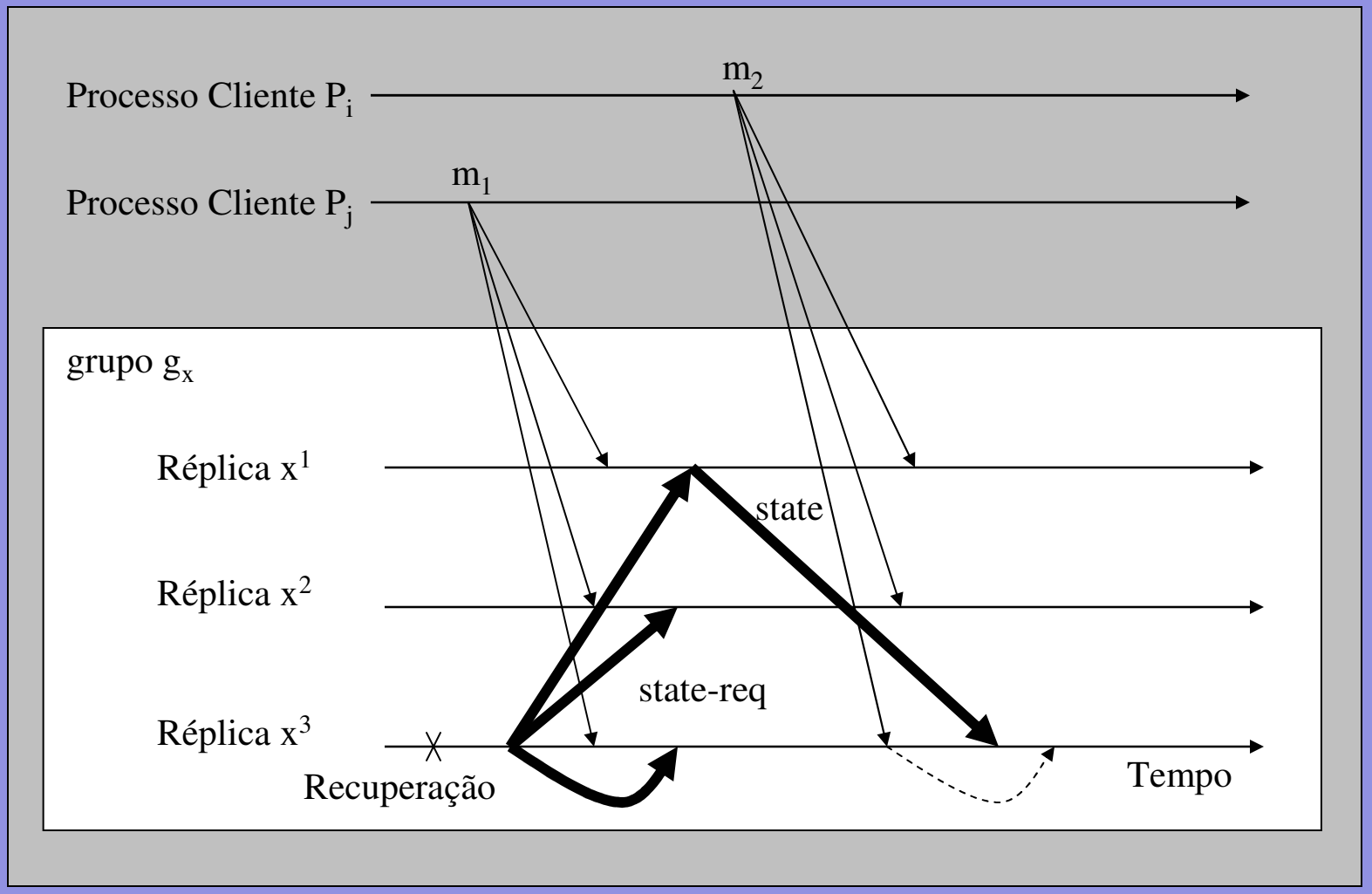
CG e Replicação ativa

- Replicação ativa requer primitiva de comunic. com as seguintes propriedades*:
  - **ordem total -** se duas réplicas entregam *m1* e *m2*, elas entreguam as mensagens na mesma ordem;
  - **atomicidade** - todas ou nenhuma das corretas recebem a mensagem; e
  - **terminação** - todas réplicas corretas *eventualmente* entregarão a mensagem.

* considerando entrega e não recebimento

- Desejável que réplicas ao recuperarem-se recebam uma transferência de estado.

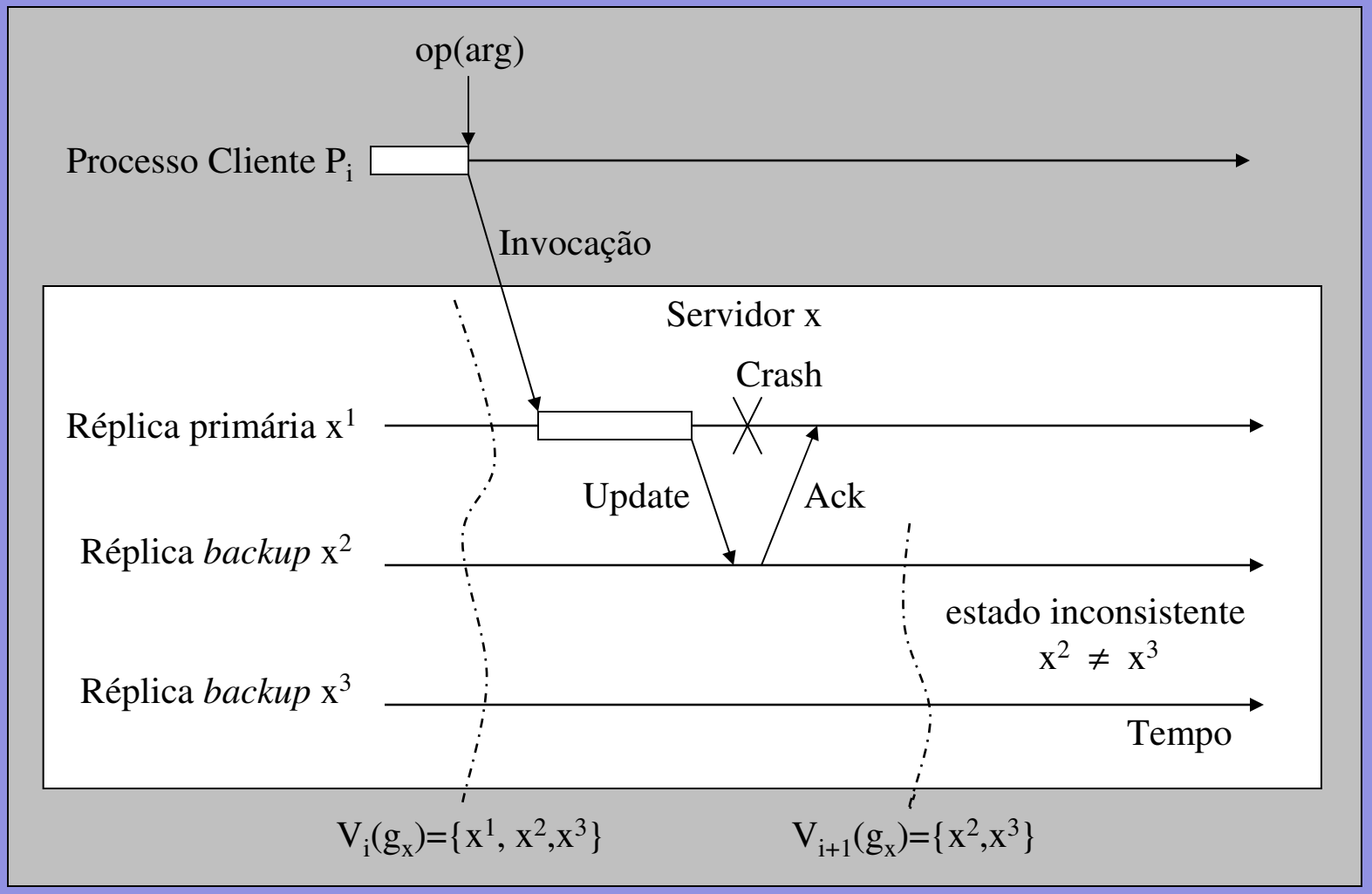
Transferência de estado



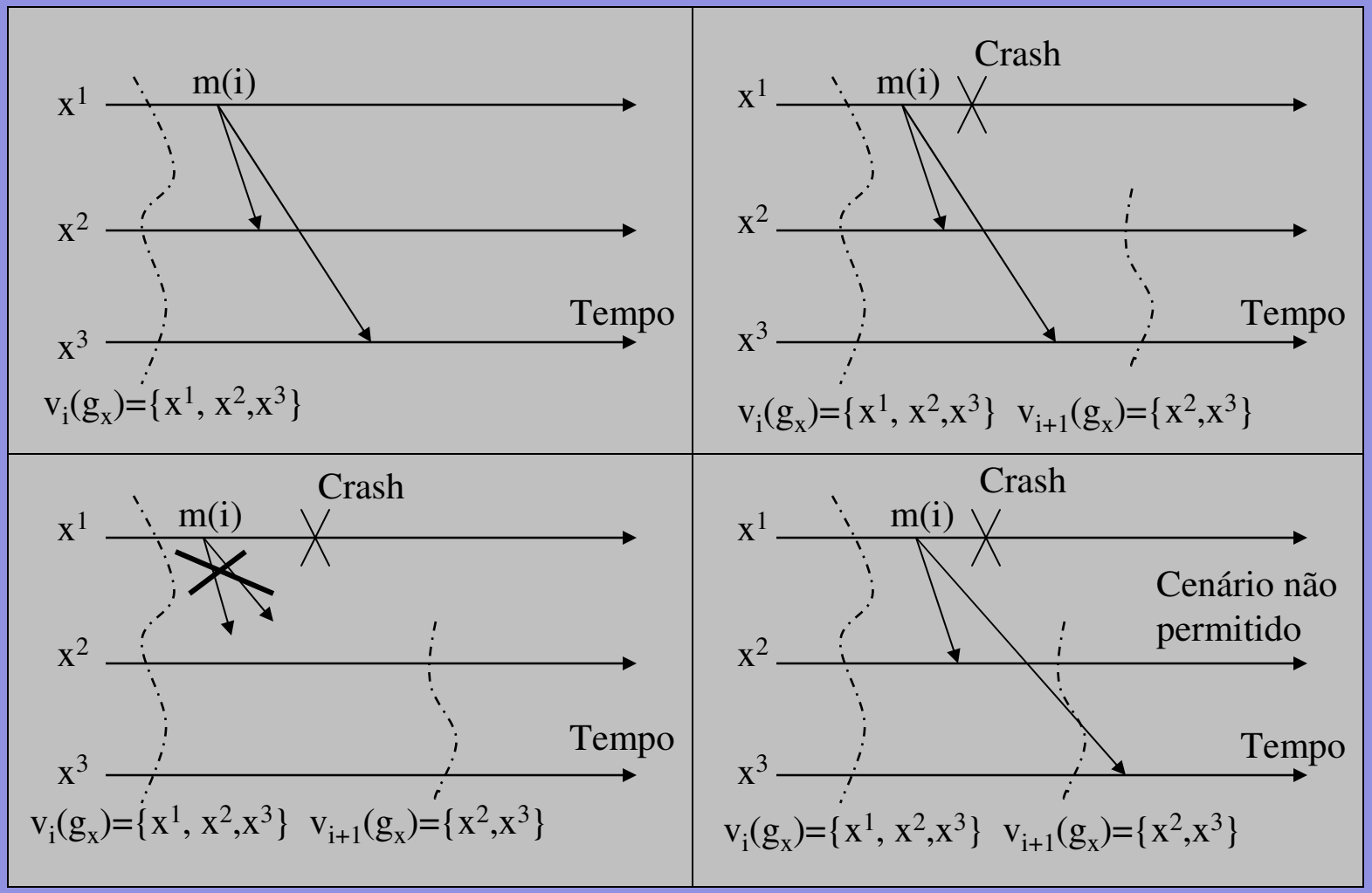
CG e Replicação *primary-backup*

- RPB não exige TOCAST
- RPB exige grupos dinâmicos
- transferência de estado pelo *join()*
- RPB exige comunicação atômica (*updates*)
- *multicast* com visão síncrona (VSCAST) fornece semântica de atomicidade

Problema se não atômico



Multicast com visão síncrona



CG e Consenso

- TOCAST $\leftrightarrow$ consenso
- VSCAST $\leftrightarrow$ consenso

TOCAST $\leftrightarrow$ consenso

- *multicast* não ordenado de *m* identificado pelo inteiro *k* (instância *k* do consenso)
- p_i recebe e coloca num *buffer* ($undeliv_i$)
- p_i inicia um alg. de consenso para o *batch(k)* de mensagens
- obtido consenso *k*, p_i entrega *batch(k)* e retira-o do *buffer*
- repete até *buffer* estar vazio

TOCAST $\leftrightarrow$ consenso

- cada $p_i \in g_x$ entrega m na seguinte ordem:

  - msg de *batch(k)* são entregues antes das do *batch(k+1)*

  - $\forall k$ as msg do *batch(k)* são entregues numa ordem determinística (ex: ordem definida pelos seus identificadores)

VSCAST $\leftrightarrow$ consenso

- alg. de consenso serve para *batch(k)* e também p/ estabelecer $v_{i+1}(g_x)$
- obtido consenso k, p_i entrega *batch(k)* e após entrega $v_{i+1}(g_x)$
- o consenso k é resolvido ou entre os processos da visão inicial $v_0(g_x)$, ou entre os processos da visão corrente $v_k(g_x)$

VSCAST $\leftrightarrow$ consenso

- instâncias do consenso baseiam-se em mensagens estáveis ($stable_i(m)$)
- $stable_i(m)$ é *true* sse p_i sabe que todos os processos em $v_k(g_x)$ receberam m
- se $p_i \in v_k(g_x)$ e recebeu m, mas $stable_i(m)$ não válida após um *timeout*, então p_i *multicast req-view(k+1)* à $v_k(g_x)$. Cada $p_j \in v_k(g_x)$ retorna por *multicast* suas mensagens não estáveis.

CG e Consenso (cont.)

- TOCAST↔consenso e VSCAST↔consenso
- maioria implementações negligencia a certeza / sobrevivência (*liviness*), ou seja, não especificam bem situações de término do algoritmo
- Certeza é difícil de oferecer devido ao iFLP (Fischer, Lynch e Paterson)
- *timeouts* são insuficientes para contornar a iFLP, necessita-se suposições adicionais

Consenso

- Consenso => processos do grupo *G* devem decidir *v* (valor comum) respeitando:
- Acordo (*Agreement*) - dois processos corretos nunca decidem adotar valores diferentes
- Validade - se um processo decide *v*, então algum processo propôs *v*
- Terminação - cada processo correto *eventualmente* toma uma decisão
- Acordo uniforme - dois processos (corretos ou não) nunca decidem adotar valores diferentes

Detetores de defeitos (DD)

Chandra e Toueg

- Consideram propriedades de *completeness* (condição de detecção) e *accuracy* (restringe erros que DD pode fazer)
- Várias classes de detetores de defeitos não confiáveis
- Possibilitam resolver consenso com maioria de corretos
- Modelo *attach* um DD em cada processo
- Cada DD mantém uma lista de suspeitos
- Suspeitos detectados por *timeout*
- Shiper usa classe ◆S:

  strong completeness/eventual weak accuracy

  Strong (3 fases) Weak (2 fases)

CG e Consenso (cont.)

- Detetores de defeito => especificam condições mínimas de garantia de término
- TOCAST e VSCAST em sist. assíncronos só são possíveis com detetores de defeitos não confiáveis

Conclusões

- TOCAST *x* replicação ativa
- VSCAST *x* replicação *primary-backup*
- Dificuldade de implementação > iFLP
- *Framework* > DD ñ confiáveis p/ sist. assín.
  - permite implem. TOCASC e VSCAST
- Muitas plataformas de suporte
  - Ex: Jgroups