ACH 2147 — Desenvolvimento de Sistemas de Informação Distribuídos

Aula 26: Tolerância a Falhas (parte 4)

Prof. Renan Alves

Escola de Artes, Ciências e Humanidades — EACH — USP

14/06/2024

Tolerância a falhas de comunicação

Falhas de comunicação

- A falha pode ocorrer no canal de comunicação em si, em vez de ocorrer nos processos que compõe o sistema
- Falhas de crash (o canal se torna indisponível)
- Falhas de omissão (a mensagem não chega)
- Falhas arbitrárias (mensagens duplicadas)

TCP

- Mascara as falhas de omissão e temporização (mensagens fora de ordem)
- Não mascara falhas de crash: é preciso refazer a conexão

Chamadas de procedimento remoto (RPC) confiáveis

O que pode dar errado?

1. O cliente não consegue localizar o servidor.
2. A mensagem de solicitação do cliente para o servidor é perdida.
3. O servidor falha após receber uma solicitação.
4. A mensagem de resposta do servidor para o cliente é perdida.
5. O cliente falha após enviar uma solicitação.

Chamadas de procedimento remoto (RPC) confiáveis

O que pode dar errado?

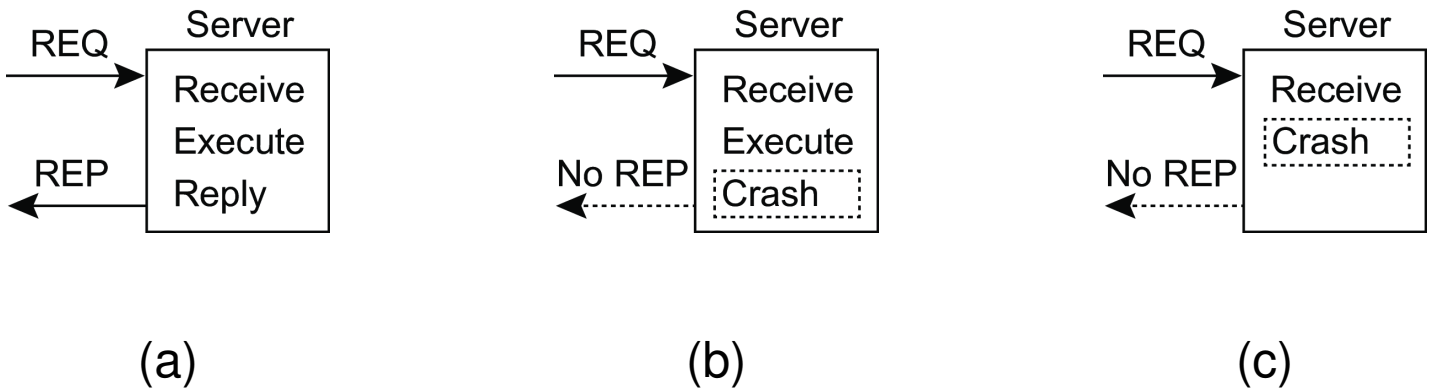
1. O cliente não consegue localizar o servidor.
2. A mensagem de solicitação do cliente para o servidor é perdida.
3. O servidor falha após receber uma solicitação.
4. A mensagem de resposta do servidor para o cliente é perdida.
5. O cliente falha após enviar uma solicitação.

Duas soluções “fáceis”

1: Cliente não consegue localizar o servidor: apenas reportar ao cliente

2: Solicitação foi perdida: apenas reenviar a mensagem

RPC Confiável: falha do servidor

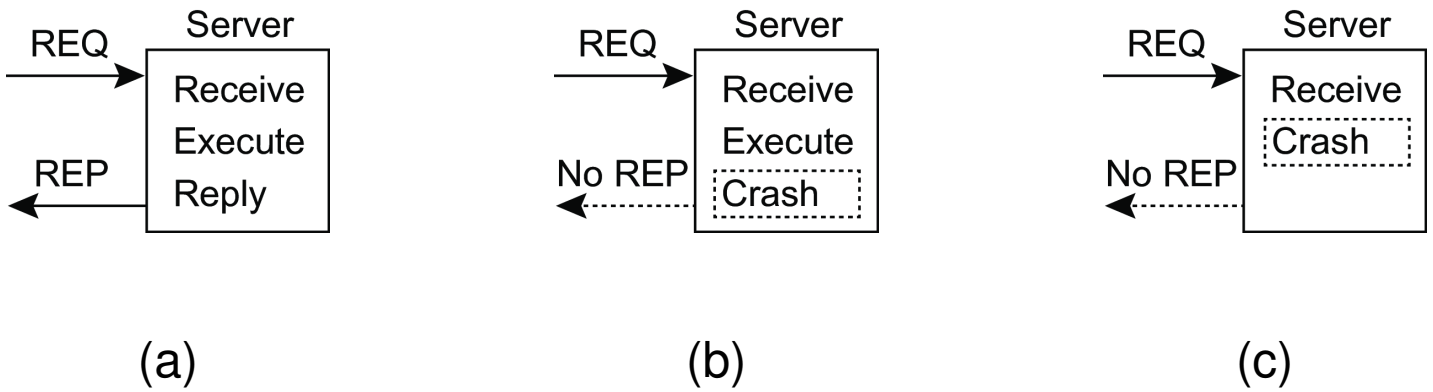


Problema

Sendo (a) é o caso normal, as situações (b) e (c) requerem soluções diferentes. No entanto, o cliente não sabe o que aconteceu. Duas abordagens:

- Semântica de pelo menos uma vez: garantia de que a operação é executada pelo menos uma vez, não importa o que aconteça.
- Semântica de no máximo uma vez: garantia de que a operação será executada no máximo uma vez.

RPC Confiável: falha do servidor



Problema

Sendo (a) é o caso normal, as situações (b) e (c) requerem soluções diferentes. No entanto, o cliente não sabe o que aconteceu. Duas abordagens:

- Semântica de pelo menos uma vez: garantia de que a operação é executada pelo menos uma vez, não importa o que aconteça.
- Semântica de no máximo uma vez: garantia de que a operação será executada no máximo uma vez.

Ideal impossível

O ideal seria um semântica exatamente uma vez, porém não é alcançável apenas com mensagens de ACK.

Por que a recuperação totalmente transparente do servidor é impossível

Três tipos de eventos no servidor

(Assumindo que uma atualização de documento foi solicitada ao servidor.)

- M: enviar a **m**ensagem de conclusão
- P: concluir o **p**rocessamento do documento
- C: falha (**c**rash)

Seis ordens possíveis

(Ações entre parênteses nunca ocorrem)

1. $M \rightarrow P \rightarrow C$: Falha após relatar conclusão.
2. $M \rightarrow C(\rightarrow P)$: Falha após relatar conclusão, mas antes da atualização.
3. $P \rightarrow M \rightarrow C$: Falha após relatar conclusão e após a atualização.
4. $P \rightarrow C(\rightarrow M)$: A atualização ocorreu e, em seguida, a falha.
5. $C(\rightarrow P \rightarrow M)$: Falha antes de fazer qualquer coisa
6. $C(\rightarrow M \rightarrow P)$: Falha antes de fazer qualquer coisa

Por que a recuperação totalmente transparente do servidor é impossível

Reissue strategy
Always
Never
Only when ACKed
Only when not ACKed
Client

Strategy M → P		
MPC	**MC(P)**	**C(MP)**
DUP	OK	OK
OK	ZERO	ZERO
DUP	OK	ZERO
OK	ZERO	OK
Server		

Strategy P → M		
PMC	**PC(M)**	**C(PM)**
DUP	DUP	OK
OK	OK	ZERO
DUP	OK	ZERO
OK	DUP	OK
Server		

OK = Document processed once
DUP = Document processed twice
ZERO = Document not processed at all

RPC Confiável: mensagens de resposta perdidas

O problema

O que o cliente percebe é que não está recebendo uma resposta. No entanto, ele não pode decidir se isso é causado por uma solicitação perdida, um servidor em falha ou uma resposta perdida.

Solução parcial

Projetar o servidor de forma que suas operações sejam idempotentes: repetir a mesma operação é o mesmo que executá-la exatamente uma vez:

- operações de leitura pura
- operações de sobrescrita estrita

Muitas operações são inerentemente não idempotentes, como muitas transações bancárias.

RPC Confiável: mensagens de resposta perdidas

O problema

O que o cliente percebe é que não está recebendo uma resposta. No entanto, ele não pode decidir se isso é causado por uma solicitação perdida, um servidor em falha ou uma resposta perdida.

Solução parcial

Projetar o servidor de forma que suas operações sejam idempotentes: repetir a mesma operação é o mesmo que executá-la exatamente uma vez:

- operações de leitura pura
- operações de sobrescrita estrita

Muitas operações são inerentemente não idempotentes, como muitas transações bancárias.

Alternativa

Usar números de sequência, porém servidor deve manter estado por cada cliente de forma persistente e atômica.

RPC Confiável: falha do cliente

Problema

O servidor está trabalhando e usando recursos para nada (computação órfã).

Ideias de solução (nenhuma é muito boa)

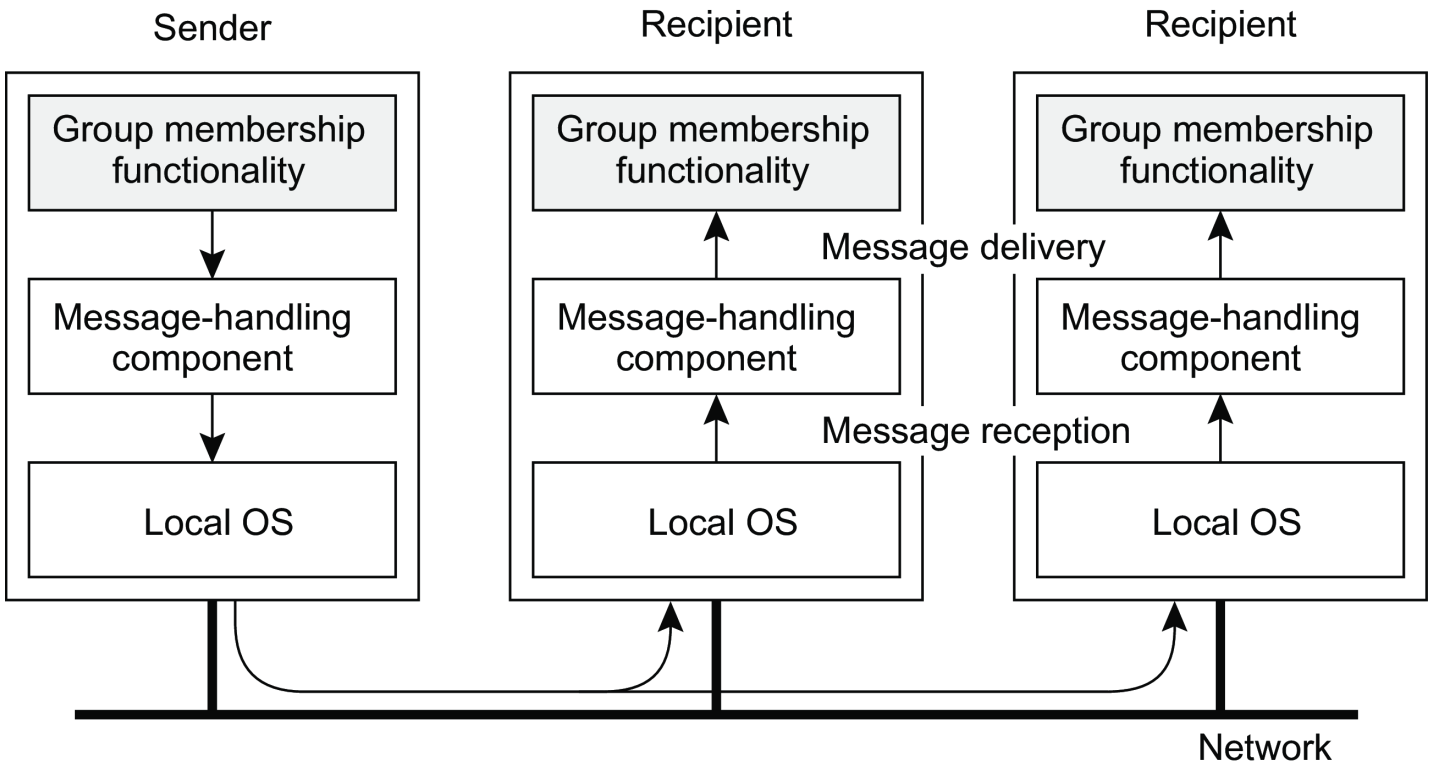
- Eliminação da computação órfã: cliente cancela o pedido quando ele se recupera
- Cliente transmite novo número de época quando se recupera $\Rightarrow$ servidor elimina qualquer computação órfã de épocas anteriores do cliente
- Exigir que as computações se completem em até T unidades de tempo. Se precisar de mais tempo, deve pedir mais ao cliente. Se não for possível contatar, o pedido é removido.

Comunicação confiável em grupo simples

Intuição

Uma mensagem enviada para um grupo de processos **G** deve ser entregue a cada membro de **G**.

Há uma diferença entre recebimento e entrega de mensagens.



Comunicação confiável em grupo menos simples

Comunicação confiável na presença de processos que podem falhar

A comunicação em grupo é confiável quando pode ser garantido que uma mensagem é recebida e subsequentemente entregue por todos os membros do grupo não falhos.

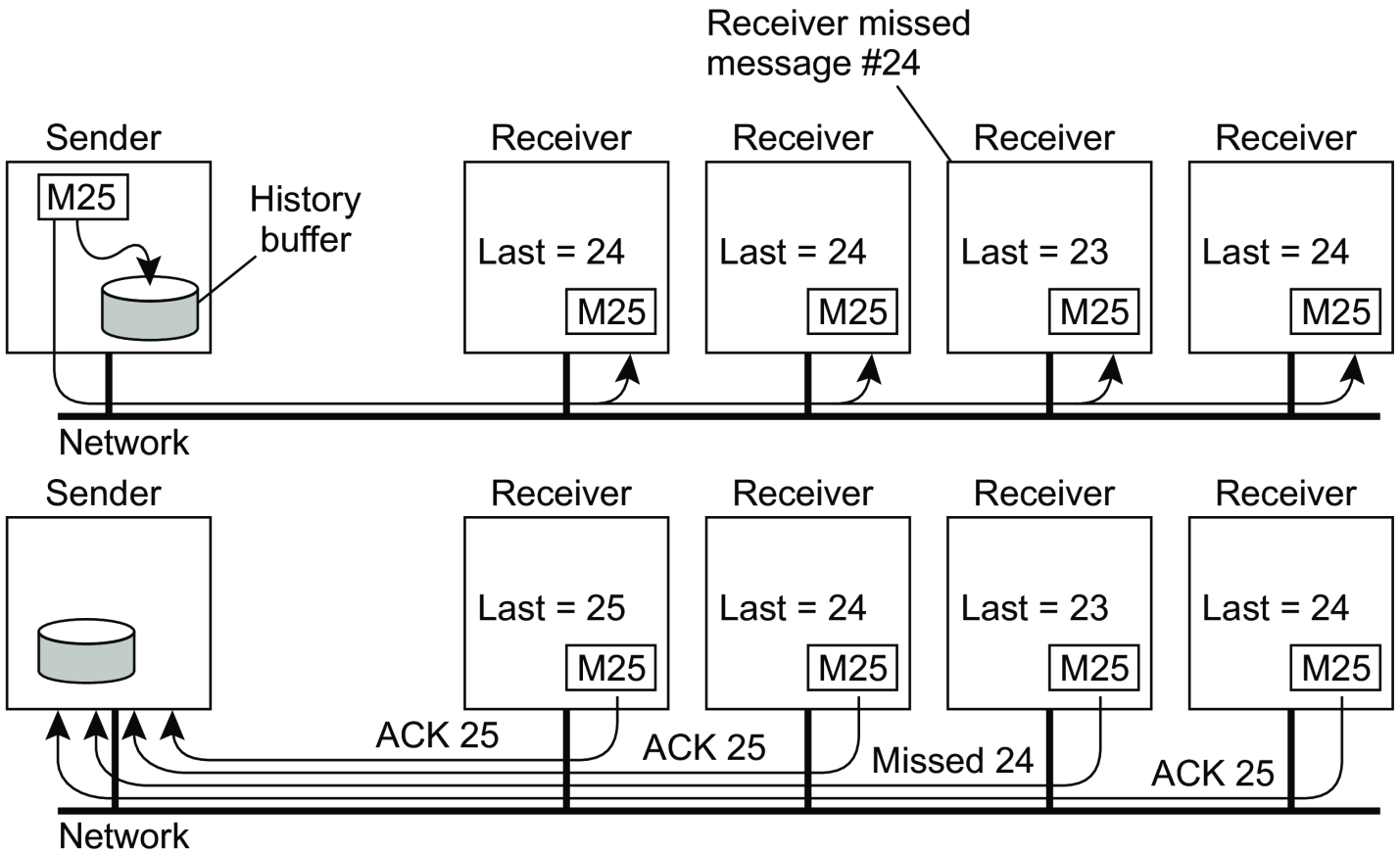
Parte complicada

É necessário acordo sobre como o estado atual do grupo antes que uma mensagem recebida possa ser entregue.

Comunicação confiável em grupo simples

Comunicação confiável, mas assumindo processos não falhos

A comunicação confiável em grupo agora se resume a multicast confiável: uma mensagem é recebida e entregue a cada destinatário, como pretendido pelo remetente.



Escalabilidade no multicast confiável

Problema

Grupos grandes resultará em muitos ACKs por mensagem

Possível solução

Usar NACKS escala melhor

- Porém ainda tem problemas:
- Ainda pode haver muitos NACKS
- Quando retirar mensagem do buffer?

Protocolo Scalable Reliable Multicasting (SRM)

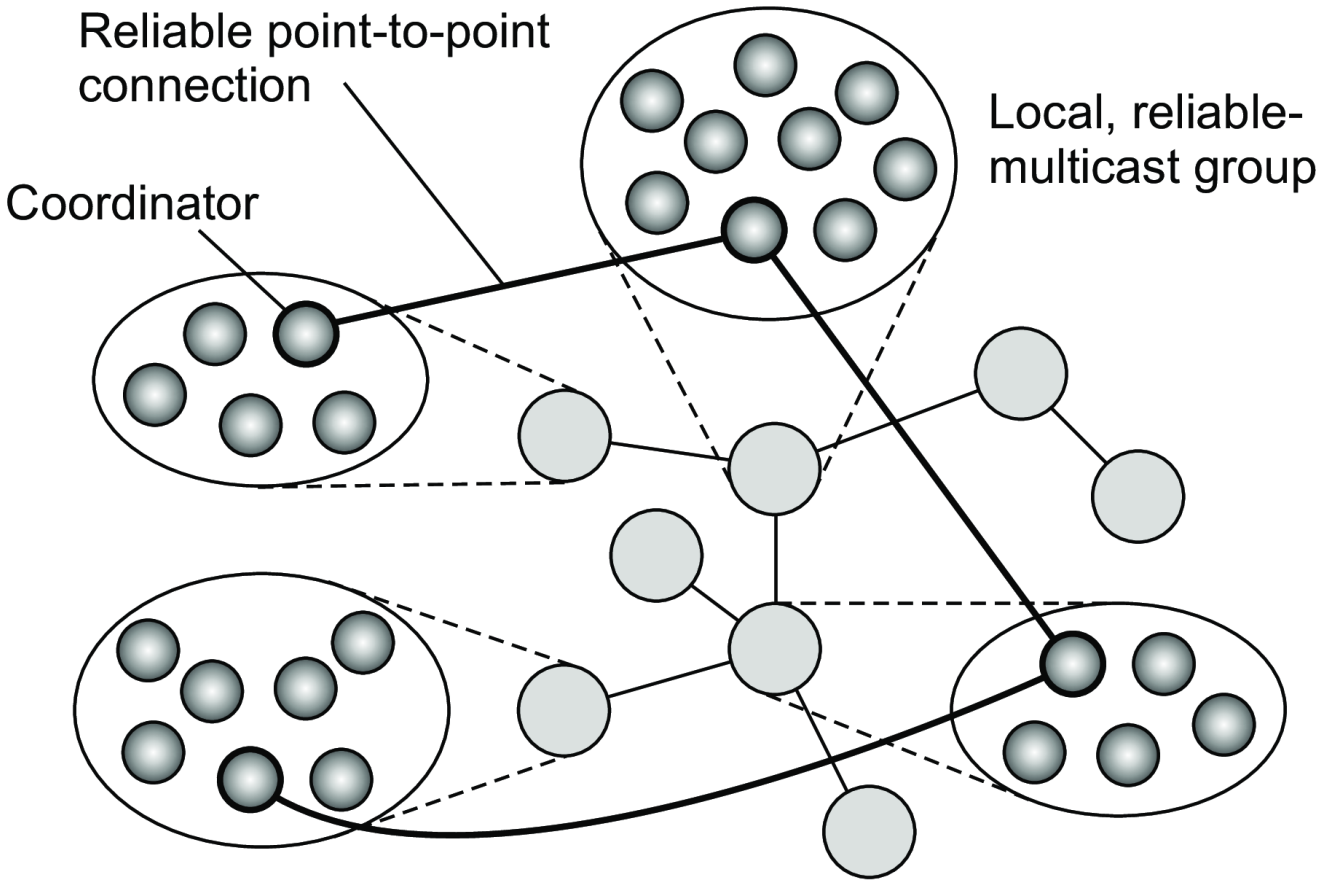
Ideia geral

- Uso de NACK
- Reenvios são feitos para todo o grupo
- NACK também é enviado para todo o grupo
- Ao receber um NACK, processe que estava pensando em enviar um NACK deixa de enviar
- Tempo aleatório para enviar NACK

Não é tão escalável assim

Processos que receberam a mensagem com sucesso recebem NACKS que não úteis para nada.

Se houver muitos processos: solução hierárquica



Multicast atômico

Ideia geral

- Ou a mensagem é entregue a todos os processos ou não é entregue a nenhum processo (Obs. a mensagem pode ser recebida, mas não entregue).
- Quando ocorre um crash de processo, é preciso atualizar a visão dos membros em funcionamento do grupo

Tipos de ordenação:

Multicast	**Basic message ordering**	**TO delivery?**
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Protocolos de commit distribuído

Problema

Ter uma operação sendo realizada por cada membro de um grupo de processos, ou nenhum.

- Multicast confiável: uma mensagem deve ser entregue a todos os destinatários.
- Transação distribuída: cada transação local deve ser bem-sucedida.

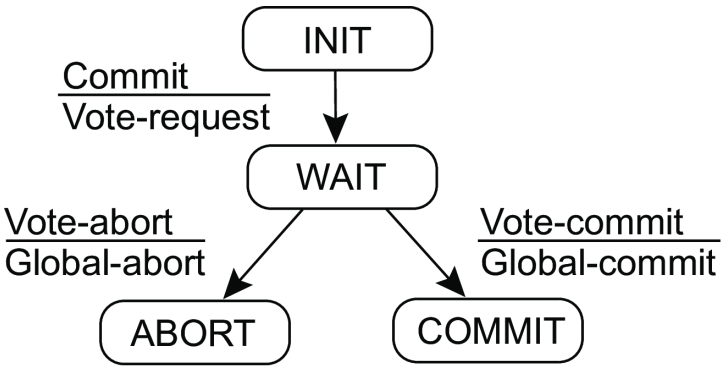
Protocolo de compromisso em duas fases (2PC)

Essência

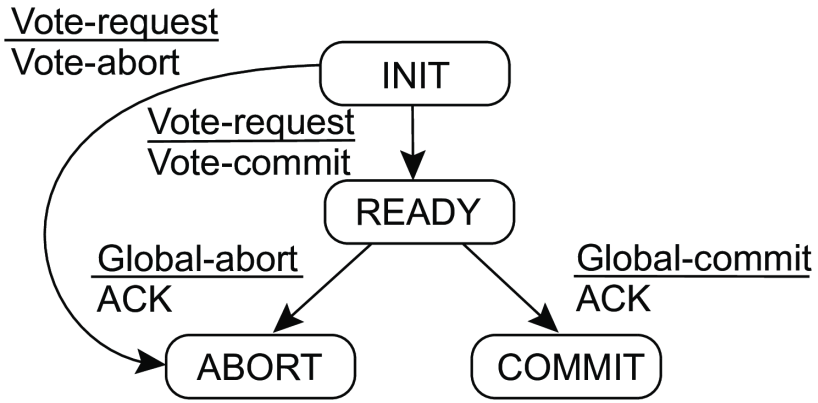
O cliente que iniciou a computação atua como coordenador; os outros processos que farão o commit são os participantes.

- Fase 1a: Coordenador envia VOTE-REQUEST para os participantes (também chamado de pré-escrita)
- Fase 1b: Quando o participante recebe VOTE-REQUEST, retorna VOTE-COMMIT ou VOTE-ABORT ao coordenador. Se enviar VOTE-ABORT, aborta sua computação local
- Fase 2a: Coordenador coleta todos os votos; se todos forem VOTE-COMMIT, envia GLOBAL-COMMIT a todos os participantes, caso contrário, envia GLOBAL-ABORT
- Fase 2b: Cada participante espera por GLOBAL-COMMIT ou GLOBAL-ABORT e lida com a situação de acordo.

2PC - Máquinas de estados finitos



Coordenador



Participante

2PC – Falha do participante

Análise: participante falha no estado S, e se recupera para S

- *INIT*: Sem problema: o participante desconhecia o protocolo

2PC – Falha do participante

Análise: participante falha no estado S, e se recupera para S

- *READY*: O participante está esperando para confirmar ou abortar. Após a recuperação, o participante precisa saber qual transição de estado deve fazer

2PC – Falha do participante

Análise: participante falha no estado *S*, e se recupera para *S*

- *ABORT*: Apenas tornar a entrada no estado de abortamento idempotente, por exemplo, removendo o estado referente à requisição

2PC – Falha do participante

Análise: participante falha no estado S, e se recupera para S

- *COMMIT*: Também tornar a entrada no estado de confirmação idempotente, por exemplo, copiando o resultado para o armazenamento final.

2PC – Falha do participante

Análise: participante falha no estado S, e se recupera para S

- *INIT*: Sem problema: o participante desconhecia o protocolo
- *READY*: O participante está esperando para confirmar ou abortar. Após a recuperação, o participante precisa saber qual transição de estado deve fazer
- *ABORT*: Apenas tornar a entrada no estado de abortamento idempotente, por exemplo, removendo o estado referente à requisição
- *COMMIT*: Também tornar a entrada no estado de confirmação idempotente, por exemplo, copiando o resultado para o armazenamento final.

Observação

Quando o commit distribuído é necessário, ter participantes que usam espaços de trabalho temporários para manter seus resultados permite uma recuperação simples na presença de falhas.

2PC – Falha do participante

Alternativa

Quando uma recuperação é necessária para o estado *READY*, verificar o estado de outros participantes $\Rightarrow$ não é necessário esperar pela decisão do coordenador.

Participante recuperado *P* contata outro participante *Q*

Estado de *Q*	**Ação de** *P*
COMMIT	Fazer transição para *COMMIT*
ABORT	Fazer transição para *ABORT*
INIT	Fazer transição para *ABORT*
READY	Contatar outro participante

Resultado

Se todos os participantes estiverem no estado *READY*, o protocolo bloqueia. Aparentemente, o coordenador está falhando. Nota: O protocolo presume que precisamos da decisão do coordenador.

2PC – Falha do coordenador

Observação

O problema está no fato de que a decisão final do coordenador pode não estar disponível por algum tempo (ou realmente perdida).

Alternativa

Permitir que um participante *P* no estado *READY* espere por um timeout quando não recebeu a decisão do coordenador; Após o timeout, *P* tenta descobrir o que outros participantes sabem (conforme discutido anteriormente).

Observação

A essência do problema é que um participante em recuperação não pode tomar uma decisão local: é dependente de outros (possivelmente falhos) processos

Recuperação: Contexto

Essência

Quando uma falha ocorre, precisamos trazer o sistema para um estado livre de erros:

- Recuperação de erro retroativa: Trazer o sistema de volta a um estado anterior livre de erros
- Recuperação de erro para frente: Encontrar um novo estado a partir do qual o sistema pode continuar operando

Prática

Use recuperação de erro retroativa, exigindo que estabeleçamos pontos de recuperação. É um mecanismo mais genérico.

Observação

A recuperação em sistemas distribuídos é complicada pelo fato de que os processos precisam cooperar para identificar um estado consistente a partir do qual se recuperar

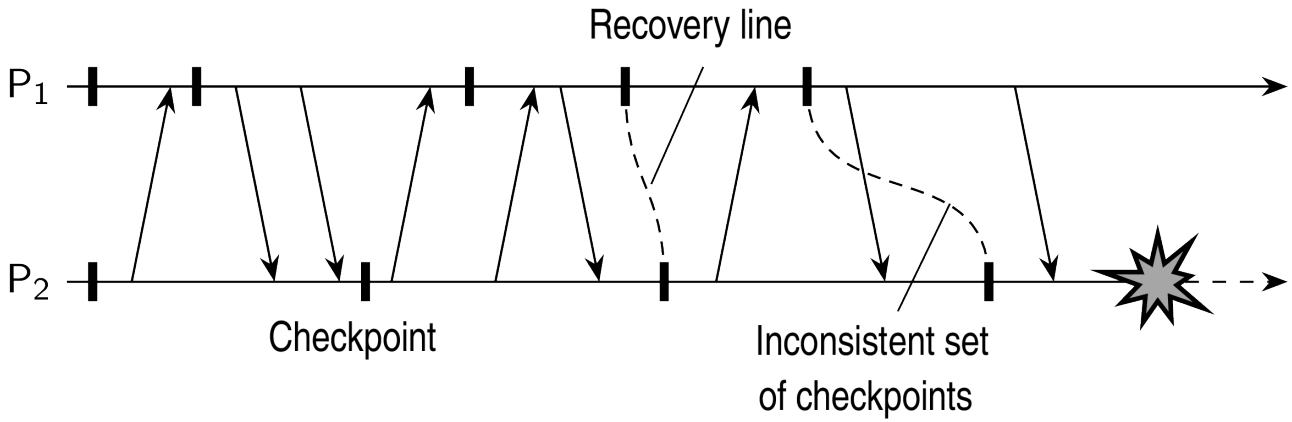
Recuperação de estado global

Requisito

Toda mensagem que foi recebida também deve ser mostrada como enviada no estado do remetente.

Linha de recuperação

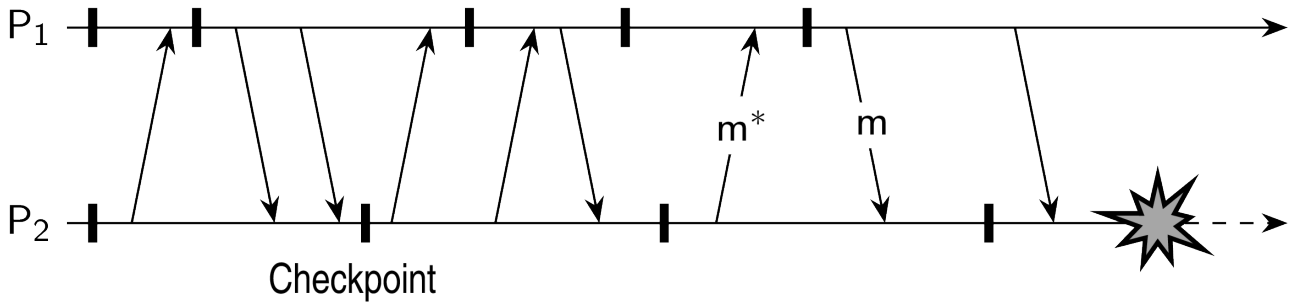
Supondo que todos os processos façam checkpoints de seu estado regularmente, a ”Linha de recuperação” é o checkpoint global consistente mais recente.



Rollback em cascata

Observação

Se o checkpointing for feito nos ”momentos errados”, a linha de recuperação pode estar no momento de inicialização do sistema. Temos um chamado rollback em cascata (efeito dominó).



Checkpointing independente

Essência

Cada processo faz checkpoints independentemente, com o risco de um rollback em cascata para a inicialização do sistema.

Checkpointing independente

Essência

Cada processo faz checkpoints independentemente, com o risco de um rollback em cascata para a inicialização do sistema.

- Considere $CP_i(m)$ como o m-ésimo checkpoint do processo P_i e $INT_i(m)$ o intervalo entre $CP_i(m-1)$ e $CP_i(m)$.

Checkpointing independente

Essência

Cada processo faz checkpoints independentemente, com o risco de um rollback em cascata para a inicialização do sistema.

- Considere $CP_i(m)$ como o m-ésimo checkpoint do processo P_i e $INT_i(m)$ o intervalo entre $CP_i(m-1)$ e $CP_i(m)$.
- Quando o processo P_i envia uma mensagem no intervalo $INT_i(m)$, ele adiciona o par (i, m) à mensagem

Checkpointing independente

Essência

Cada processo faz checkpoints independentemente, com o risco de um rollback em cascata para a inicialização do sistema.

- Considere $CP_i(m)$ como o m-ésimo checkpoint do processo P_i e $INT_i(m)$ o intervalo entre $CP_i(m-1)$ e $CP_i(m)$.
- Quando o processo P_i envia uma mensagem no intervalo $INT_i(m)$, ele adiciona o par (i, m) à mensagem
- Quando o processo P_j recebe uma mensagem no intervalo $INT_j(n)$, ele registra a dependência $INT_i(m) \rightarrow INT_j(n)$.

Checkpointing independente

Essência

Cada processo faz checkpoints independentemente, com o risco de um rollback em cascata para a inicialização do sistema.

- Considere $CP_i(m)$ como o m-ésimo checkpoint do processo P_i e $INT_i(m)$ o intervalo entre $CP_i(m-1)$ e $CP_i(m)$.
- Quando o processo P_i envia uma mensagem no intervalo $INT_i(m)$, ele adiciona o par (i, m) à mensagem
- Quando o processo P_j recebe uma mensagem no intervalo $INT_j(n)$, ele registra a dependência $INT_i(m) \rightarrow INT_j(n)$.
- A dependência $INT_i(m) \rightarrow INT_j(n)$ é salva no armazenamento persistente ao fazer o checkpoint $CP_j(n)$.

Checkpointing independente

Essência

Cada processo faz checkpoints independentemente, com o risco de um rollback em cascata para a inicialização do sistema.

- Considere $CP_i(m)$ como o m-ésimo checkpoint do processo P_i e $INT_i(m)$ o intervalo entre $CP_i(m-1)$ e $CP_i(m)$.
- Quando o processo P_i envia uma mensagem no intervalo $INT_i(m)$, ele adiciona o par (i, m) à mensagem
- Quando o processo P_j recebe uma mensagem no intervalo $INT_j(n)$, ele registra a dependência $INT_i(m) \rightarrow INT_j(n)$.
- A dependência $INT_i(m) \rightarrow INT_j(n)$ é salva no armazenamento persistente ao fazer o checkpoint $CP_j(n)$.

Observação

Se o processo P_i fizer rollback para $CP_i(m-1)$, P_j deve fazer rollback para $CP_j(n-1)$. Em seguida, Deve ser verificado se o estado global é consistente.

Checkpointing coordenado

Essência

Cada processo faz um checkpoint após uma ação coordenada globalmente.

Solução simples

Usar um protocolo de bloqueio em duas fases:

Checkpointing coordenado

Essência

Cada processo faz um checkpoint após uma ação coordenada globalmente.

Solução simples

Usar um protocolo de bloqueio em duas fases:

- Um coordenador multicast uma mensagem de solicitação de checkpoint

Checkpointing coordenado

Essência

Cada processo faz um checkpoint após uma ação coordenada globalmente.

Solução simples

Usar um protocolo de bloqueio em duas fases:

- Um coordenador multicast uma mensagem de solicitação de checkpoint
- Quando um participante recebe essa mensagem, ele faz um checkpoint, para de enviar mensagens (da aplicação) e informa que fez um checkpoint

Checkpointing coordenado

Essência

Cada processo faz um checkpoint após uma ação coordenada globalmente.

Solução simples

Usar um protocolo de bloqueio em duas fases:

- Um coordenador multicast uma mensagem de solicitação de checkpoint
- Quando um participante recebe essa mensagem, ele faz um checkpoint, para de enviar mensagens (da aplicação) e informa que fez um checkpoint
- Quando todos os checkpoints forem confirmados no coordenador, ele envia uma mensagem de checkpoint concluído para permitir que todos os processos continuem

Checkpointing coordenado

Essência

Cada processo faz um checkpoint após uma ação coordenada globalmente.

Solução simples

Usar um protocolo de bloqueio em duas fases:

- Um coordenador multicast uma mensagem de solicitação de checkpoint
- Quando um participante recebe essa mensagem, ele faz um checkpoint, para de enviar mensagens (da aplicação) e informa que fez um checkpoint
- Quando todos os checkpoints forem confirmados no coordenador, ele envia uma mensagem de checkpoint concluído para permitir que todos os processos continuem

Observação: checkpoint incremental

É possível considerar apenas aqueles processos que receberam uma mensagem causalmente dependente do coordenador e ignorar o resto dos processos (feito de forma iterativa).

Registro de mensagens (logs)

Alternativa

Em vez de fazer checkpoints frequentemente (caro), é possível tentar reproduzir o estado a partir do checkpoint mais recente e do histórico de comunicação $\Rightarrow$ armazenar mensagens em um log.

Suposição

Assumimos um modelo de execução determinístico por partes:

- A execução de cada processo pode ser considerada como uma sequência de intervalos de estado
- Cada intervalo de estado começa com um evento não determinístico (por exemplo, recebimento de mensagem)
- A execução em um intervalo de estado é determinística

Conclusão

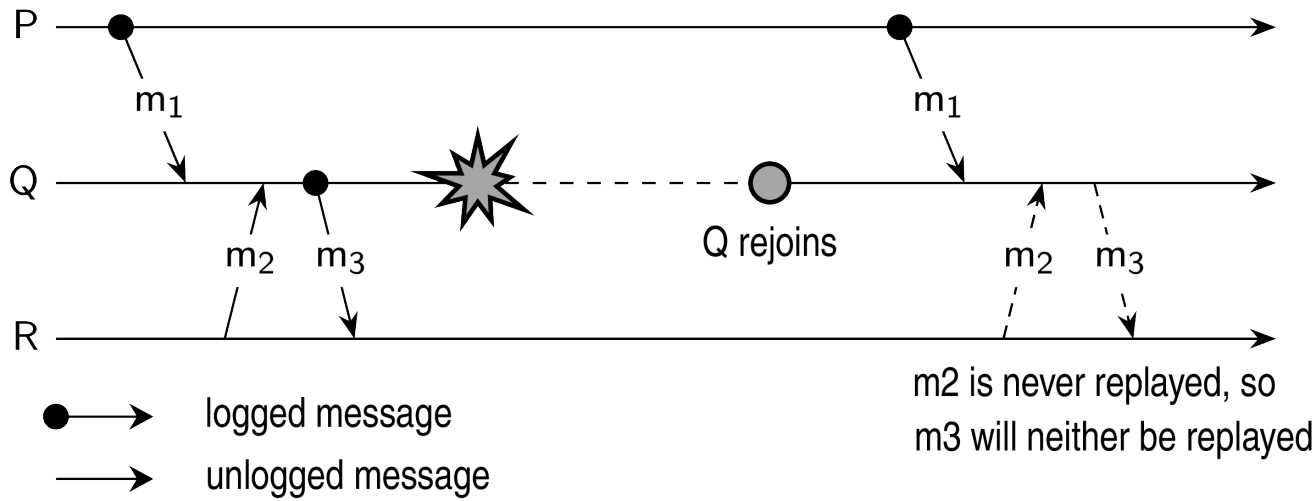
Se registrarmos eventos não determinísticos (para reproduzi-los posteriormente), obteremos um modelo de execução determinístico que nos permitirá fazer uma reprodução completa.

Registro de mensagens e consistência

Quando devemos registrar mensagens?

Para evitar processos órfãos:

- O processo Q acabou de receber e entregar as mensagens m_1 e m_2
- Suponha que m_2 nunca seja registrada.
- Após entregar m_1 e m_2, Q envia a mensagem m_3 para o processo R
- O processo R recebe e posteriormente entrega m_3: ele é um órfão.



Esquemas de registro de mensagens (log)

Notações

- **DEP**(m): processos aos quais m foi entregue. Se a mensagem m^* é causalmente dependente da entrega de m, e m^* foi entregue a Q, então $Q \in$ **DEP**(m).
- **COPY**(m): processos que têm uma cópia de m, mas ainda não a armazenaram de forma confiável.
- **FAIL**: a coleção de processos que falharam.

Caracterização

$$Q \text{ é órfão} \Leftrightarrow \exists m : Q \in \textbf{DEP}(m) \text{ e } \textbf{COPY}(m) \subseteq \textbf{FAIL}$$

Esquemas de registro de mensagens

Protocolo pessimista

Para cada mensagem não estável m, há no máximo um processo dependente de m, ou seja $|\mathbf{DEP}(m)| \leq 1$.

Consequência

Uma mensagem instável em um protocolo pessimista deve ser estabilizada antes de enviar uma próxima mensagem.

Esquemas de registro de mensagens

Protocolo otimista

Para cada mensagem instável m, garantimos que, se **COPY**$(m) \subseteq$ **FAIL**, então eventualmente também **DEP**$(m) \subseteq$ **FAIL**.

Consequência

Para garantir que **DEP**$(m) \subseteq$ **FAIL**, geralmente fazemos rollback de cada processo órfão Q até que $Q \notin$ **DEP**(m).