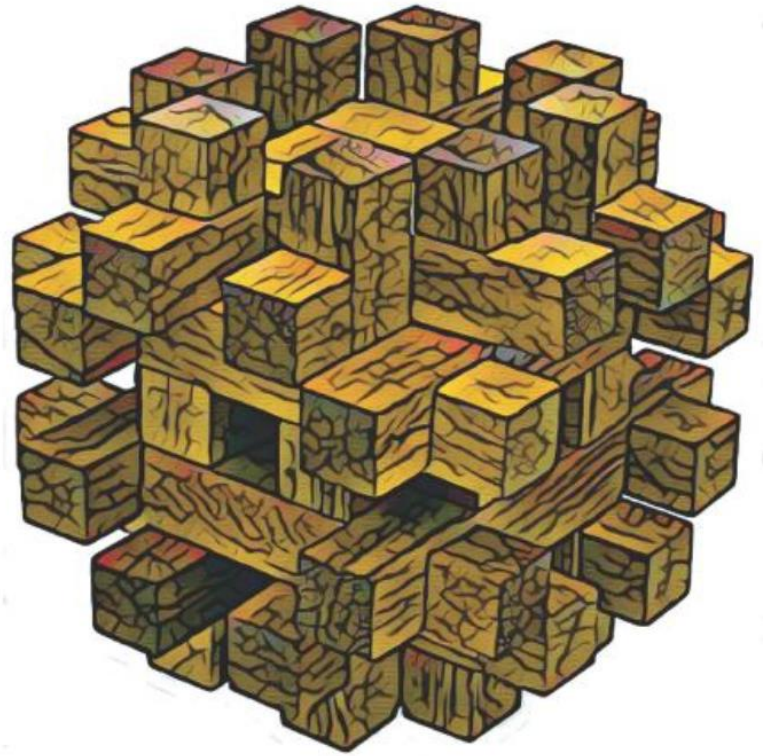
Distributed Systems

Maarten Van Steen & Andrew S. Tanenbaum



3th Edition – Version 3.03 - 2020

Capítulo 8

Tolerância a Falhas

Quinta-feira, 08 de Setembro de 2022

DEPENDABILITY (CONFIABILIDADE / DEPENDABILIDADE)

BÁSICO

Um componente provê serviços para clientes. Para prover serviços, o componente pode precisar de serviços de outros componentes => um componente pode depender em outros componentes.

ESPECIFICAMENTE

Um componente C depende em C* se a exatidão do comportamento de C depende da exatidão do comportamento de C*. (componentes são processos ou canais)

REQUISITOS RELACIONADOS A CONFIABILIDADE

Requirement	Description
Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	How easy can a failed system be repaired

CONFIABILIDADE X DISPONIBILIDADE

DISPONIBILIDADE A(t) DE UM COMPONENTE C

Fração média de tempo em que C está ativo e rodando em um intervalo [0,t]

- Disponibilidade de período longo A: A(∞)
- Nota: A = $\frac{MTTF}{MTBF} = \frac{MTTF}{MTTF+MTTR}$

OBSERVAÇÃO

Confiabilidade e disponibilidade faz sentido somente se temos uma noção precisa do que é na verdade uma falha

TERMINOLOGIA

FAILURE, ERROR, FAULT (FALHA, ERRO, CULPA)

Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

TERMINOLOGIA

TRATANDO A CULPA

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don't hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

MODELOS DE FALHAS

TIPOS DE FALHAS

Type	Description of server's behavior
Crash failure	Halts, but is working correctly until it halts
Omission failure	Fails to respond to incoming requests
Receive omission	Fails to receive incoming messages
Send omission	Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure	Response is incorrect
Value failure	The value of the response is wrong
State-transition failure	Deviates from the correct flow of control
Arbitrary failure	May produce arbitrary responses at arbitrary times

DEPENDÊNCIA X SEGURANÇA

OMISSÃO X COMISSÃO (INCUMBÊNCIA)

Falhas arbitrárias são as vezes classificadas como maliciosas. É melhor fazer a seguinte distinção:

- Falhas por omissão: um componente falha em tomar uma ação que deveria ter tomado
- Falha no comissionamento: um componente toma uma ação que ele não deveria ter tomado

OBSERVAÇÃO

Note que falhas deliberadas por omissão ou comissão são problemas típicos de segurança. A distinção entre falhas deliberadas e falhas não intencionais é em geral impossível.

PARANDO FALHAS

SUPOSIÇÕES QUE PODEMOS FAZER

Halting type	Description
Fail-stop	Crash failures, but reliably detectable
Fail-noisy	Crash failures, eventually reliably detectable
Fail-silent	Omission or crash failures: clients cannot tell what went wrong
Fail-safe	Arbitrary, yet benign failures (i.e., they cannot do any harm)
Fail-arbitrary	Arbitrary, with malicious failures

REDUNDÂNCIA POR MASCARAMENTO DE FALHAS

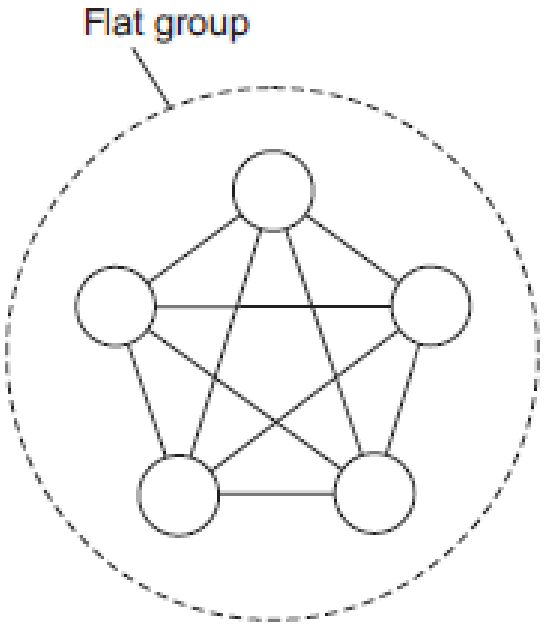
TIPOS DE REDUNDÂNCIA

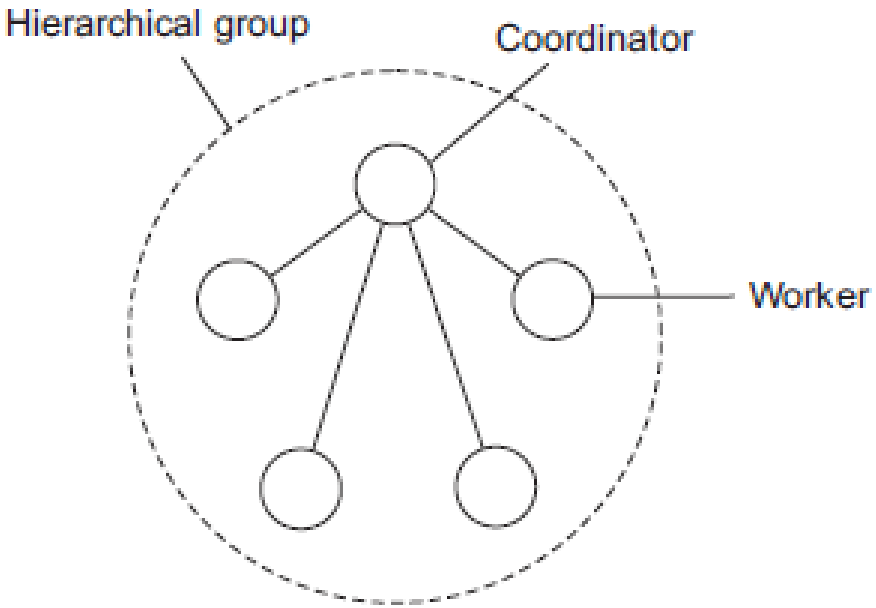
- Redundância de informação: adicione bits extra para dados de forma que erros podem ser recuperados quando os bits forem truncados
- Redundância de tempo: desenhe um sistema de forma que uma ação possa ser desempenhada novamente se alguma coisa der errado. Tipicamente usado quando falhas são transientes ou intermitentes.
- Redundância física: adicione equipamentos ou processos de forma a permitir um ou mais componentes falharem. Este tipo é extensivamente usado em sistemas distribuídos.

RESILIÊNCIA DE PROCESSOS

IDÉIA BÁSICA

Protege contra o mal funcionamento de processos através de replicação de processos, organizando processos em grupos.
Distinção entre grupos Flat e grupos hierárquicos.





GRUPOS E MASCARAMENTO DE FALHAS

GRUPOS TOLERANTES A k-ÉSIMA FALHAS

Quando um grupo pode mascarar qualquer k falhas concorrentes (k é chamado grau de tolerância a falhas)

QUÃO GRANDE DEVE SER UM GRUPO TOLERANTE A K-ÉSIMA FALHAS

- Com falhas de parada (halting) (queda / omissão / falhas de temporização): precisamos de um total de k + 1 membros dado que um não membro vai produzir um resultado incorreto, assim o resultado de um membro é bom o suficiente.
- Com falhas arbitrárias: precisamos de 2k + 1 membros de forma que o resultado correto pode ser obtido através de voto majoritário.

SUPOSIÇÕES IMPORTANTES

- Todos membros são idênticos
- Todos comandos de processos membro ocorrem na mesma ordem

Resultado: podemos ter certeza que todos processos fazem exatamente a mesma coisa.

CONSENSO :

PRÉ REQUISITO

Em um grupo de processos tolerantes a falha, cada processo sem falha executa os mesmos comandos, e na mesma ordem que outros processos sem falhas.

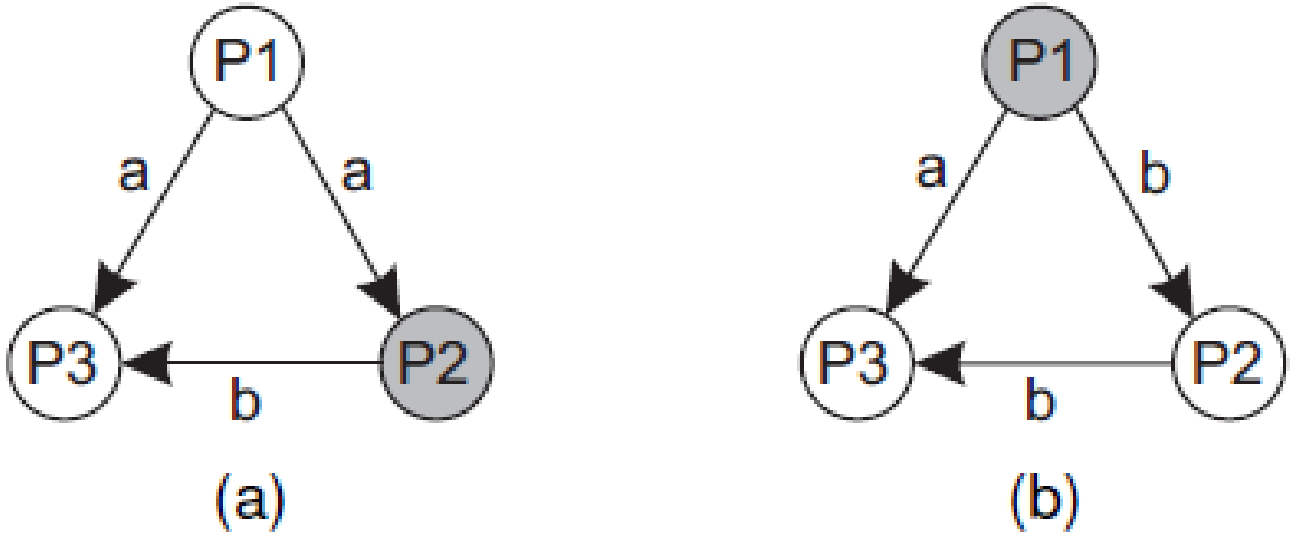
REFORMULAÇÃO

- Processos membros e um grupo sem falhas precisam chegar a um consenso sobre qual comando executar em seguida

CONSENSO EM SEMÂNTICA DE FALHAS ARBITRÁRIAS

ESSÊNCIA

Vamos considerar um grupo de processos no qual a comunicação entre processos está inconsistente: (a) repasse inapropriado de mensagens, ou (b) dizendo coisas diferentes para diferentes processos



CONSENSO EM SEMÂNTICA DE FALHAS ARBITRÁRIAS

MODELO SISTÊMICO

- Se considerarmos um repositório primário *P* e *n-1* backups *B1, ... , Bn-1*
- Um cliente envia *v ϵ {T, F}* para *P*
- Mensagens podem ser perdidas mas isto pode ser detectado
- Mensagens não podem ser corrompidas além da detecção
- Um recipiente de uma mensagem pode de forma confiável detectar seu remetente

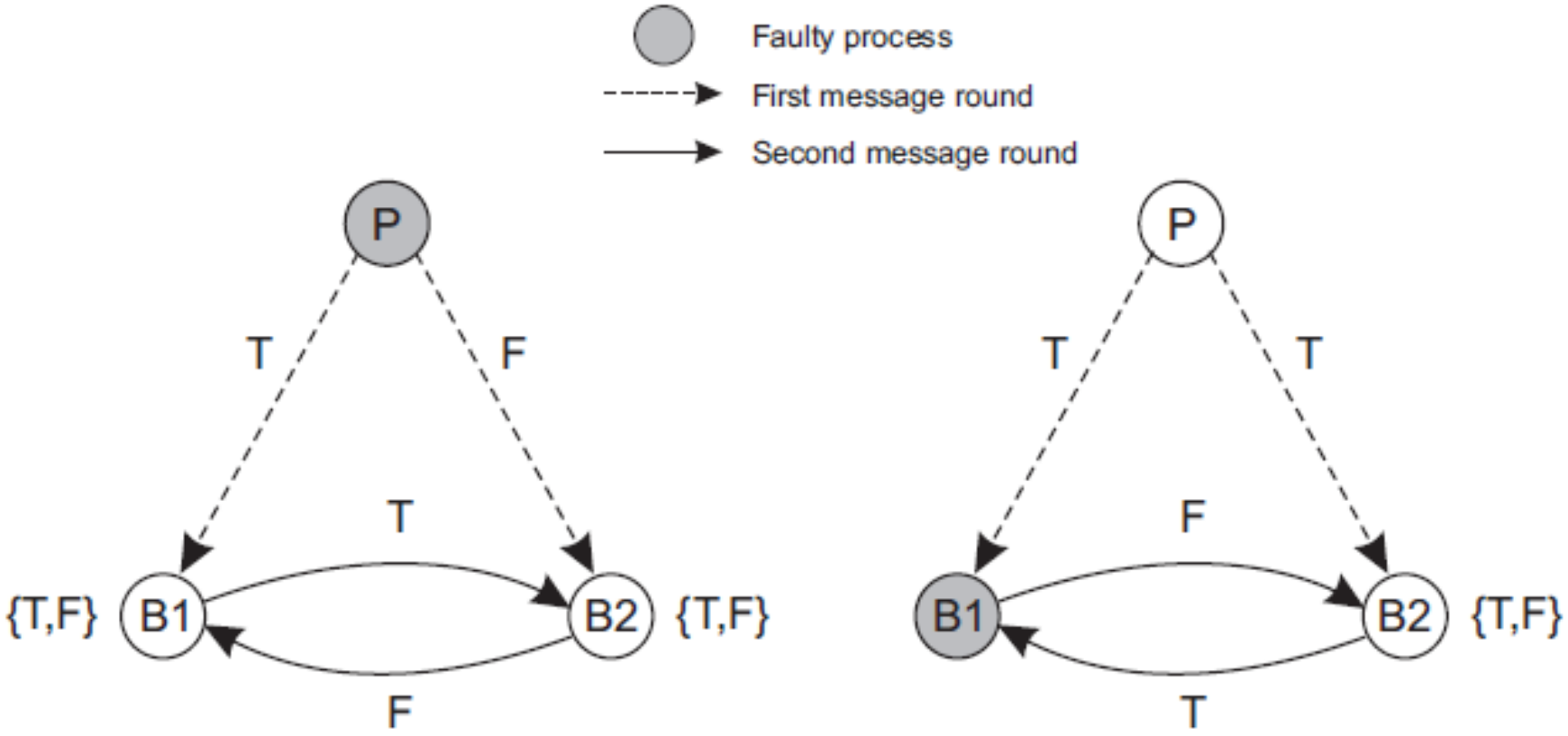
REQUISITOS PARA ACORDO BIZANTINO

BA1: cada processo de repositório backup sem falha tem o mesmo valor
BA2: se o (processo) repositório primário está sem falha então cada backup sem falha armazena exatamente o que o primário enviou

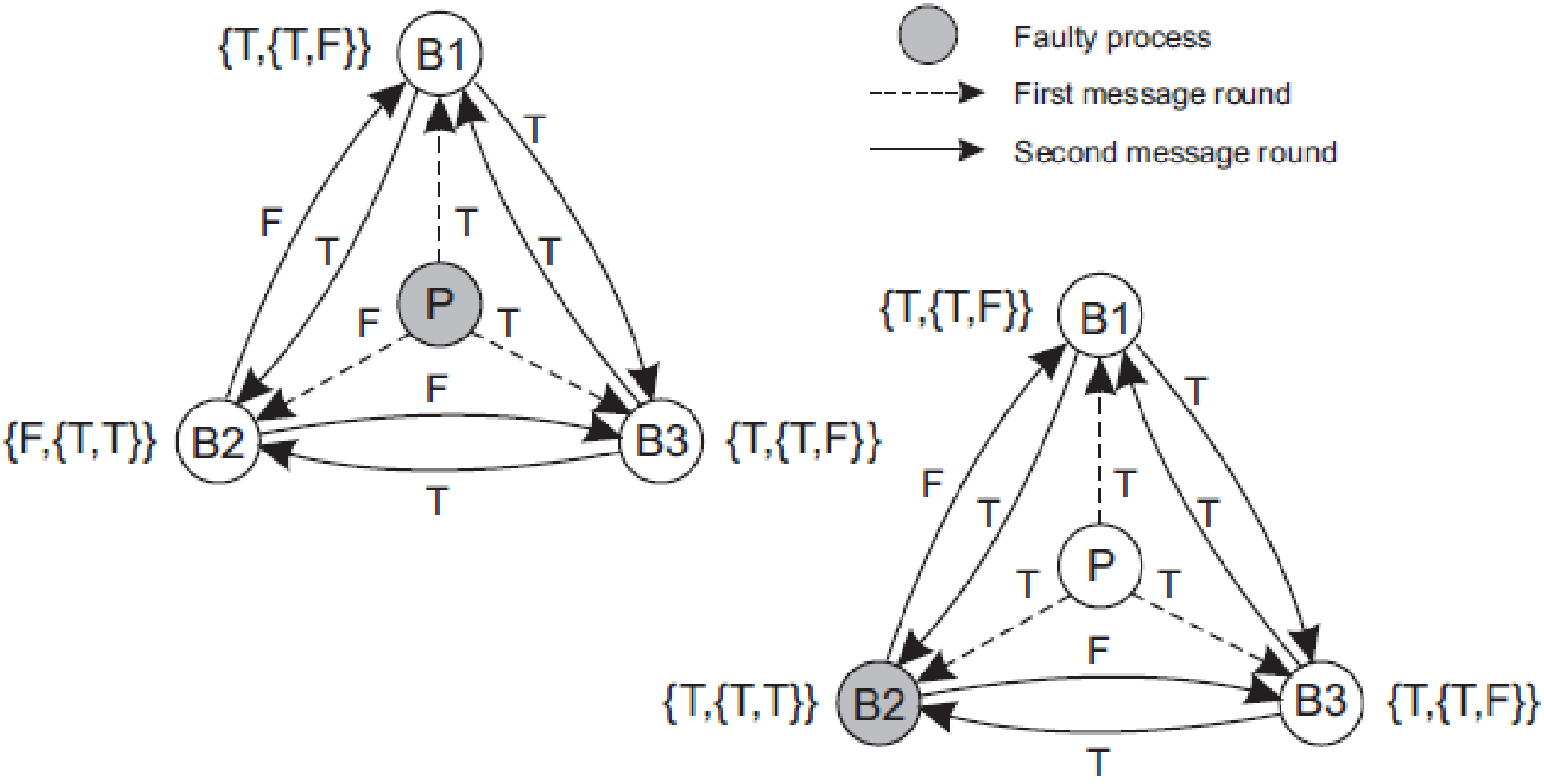
OBSERVAÇÃO

- Falha primária => BA1 fala que backups podem armazenar o mesmo, mas diferentes (e assim errado) valores do original enviados pelo cliente
- Primário sem falha => satisfaz BA2 o que implica que BA1 está satisfeito

PORQUE 3K PROCESSOS NÃO É SUFICIENTE



PORQUE 3K+1 PROCESSOS É SUFICIENTE



DETECÇÃO DE FALHAS

QUESTÃO

Podemos agora de forma confiável detectar que um processo caiu ?

MODELO GERAL

- Cada processo é equipado com um módulo de detecção de falhas
- Um processo *P* amostra outro processo *Q* e espera uma reação
- Se *Q* reagir: que é considerado estar vivo por *P*
- Se *Q* não reagir em t unidades de tempo: *Q* é suspeito de ter caído

OBSERVAÇÃO PARA UM SISTEMA SÍNCRONO

- Queda suspeita = queda conhecida

DETECÇÃO DE FALHAS NA PRATICA

Podemos agora de forma confiável detectar que um processo caiu ?

IMPLEMENTAÇÃO

- Se P não receber um heartbeat de Q dentro de tempo t: P suspeita de Q
- Se Q enviar mensagem atrasada (e for recebida por P)
  - P para de suspeitar de Q
  - P incrementa o valor de timeout de t
  - Nota: se Q caiu, P continuará suspeitando de Q

RPC CONFIÁVEL

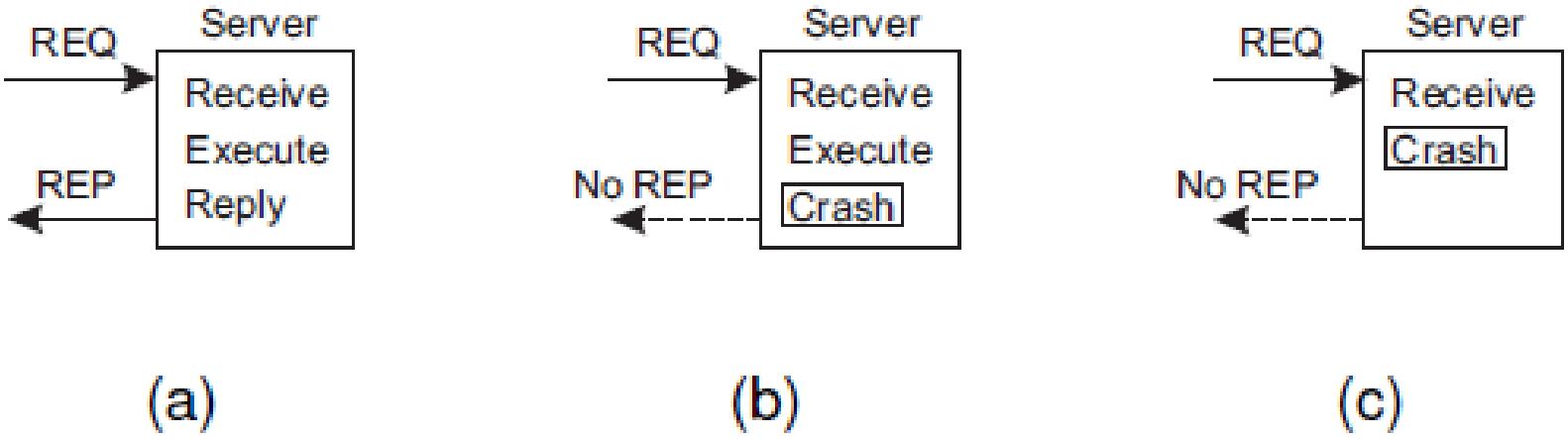
O QUE PODE DAR ERRADO

1. O cliente não consegue localizar o servidor
2. A mensagem de requisição do cliente para o servidor é perdida
3. O servidor caiu depois de receber a requisição
4. A mensagem de reply do servidor para cliente é perdida
5. O cliente caiu depois de enviar a requisição

DUAS SOLUÇÕES FÁCEIS

1. Não consegue localizar servidor: avisa o cliente
2. Requisição foi perdida: reenvie a mensagem

RPC CONFIÁVEL QUEDA DO SERVIDOR



PROBLEMA

Onde (a) é um caso normal, (b) e (c) precisam de soluções diferentes, entretanto não sabemos o que ocorreu: Duas abordagens:

- Semântica at-least-once: o servidor garante que irá executar uma operação pelo menos uma vez, independente de qualquer coisa
- Semântica at-most-once: o servidor garante que irá desempenhar uma operação no máximo uma vez

PORQUE É IMPOSSÍVEL A TRANSPARÊNCIA TOTAL "SERVER RECOVERY"

TRÊS TIPOS DE EVENTOS NO SERVIDOR

(Assuma que o servidor é requisitado atualizar um documento)

M: envia uma mensagem de finalização

P: completa o processamento de um documento

C: cai

SEIS POSSÍVEIS ORDENAMENTOS

(Ações entre parênteses nunca acontecem)

1. M → P → C: cai depois de reportar finalização
2. M → C → P: cai depois de reporta finalização, mas antes da atualização
3. P → M → C: cai depois de reportar finalização, e depois da atualização
4. P → C(→ M): atualização ocorre, e então cai
5. C (→ P → M): cai antes de fazer qualquer coisa
6. C (→ M → P):cai antes de fazer qualquer coisa

PORQUE É IMPOSSÍVEL A TRANSPARÊNCIA TOTAL "SERVER RECOVERY"

	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK
Client	Server			Server		

OK = Document updated once

DUP = Document updated twice

ZERO = Document not update at all

RPC CONFIÁVEL: MENSAGENS DE REPLY PERDIDAS

A QUESTÃO REAL

O que o cliente percebe, é que não está recebendo respostas. Entretanto ele não pode decidir se foi causado por uma mensagem perdida, queda do servidor, ou resposta perdida

SOLUÇÃO PARCIAL

Projete o servidor de forma que as operações sejam idempotentes: repetindo a mesma operação é o mesmo que executa-la exatamente uma vez:

- Operações de leitura pura
- Operações de sobrescrita estrita

Muitas operações são inerentemente não-idempotentes, tais como operações bancárias

RPC CONFIÁVEL: CLIENT CRASH

PROBLEMA

O servidor está fazendo trabalho e segurando recursos por nada (chamada de computação órfã).

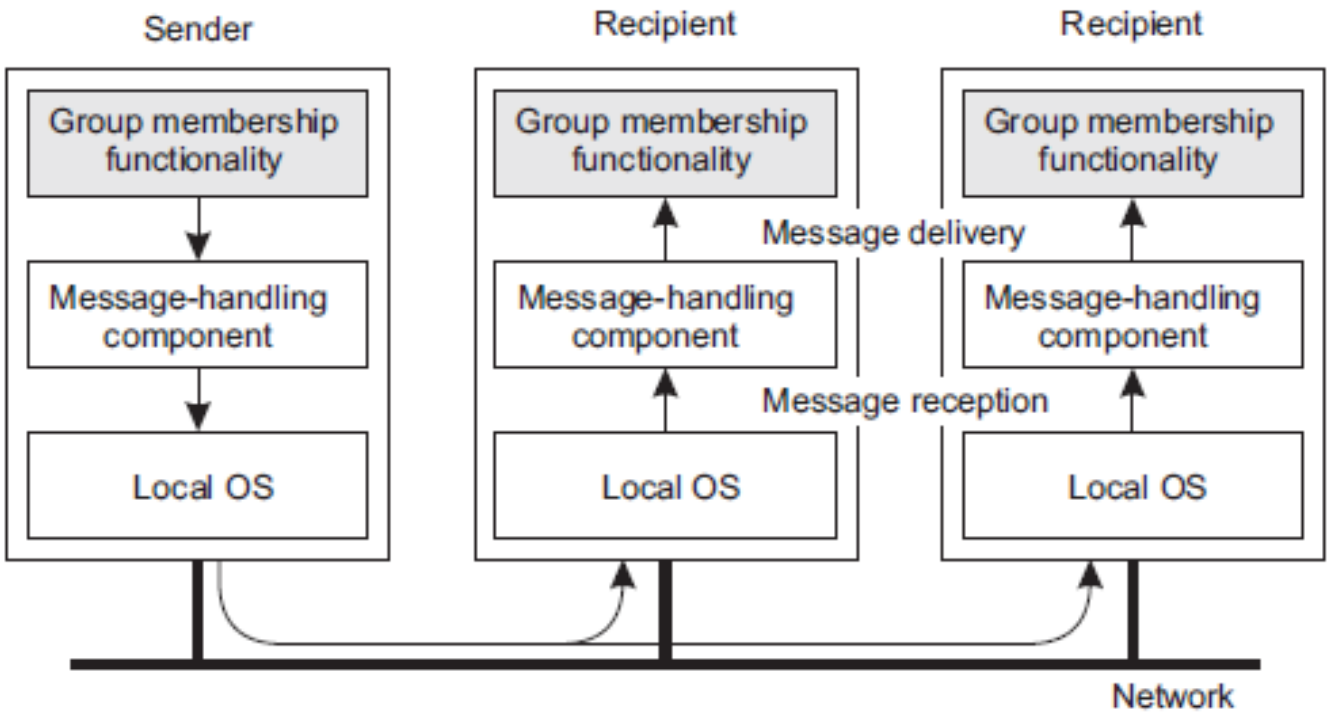
SOLUÇÃO

- Órfã é morta (killed ou rolled back) pelo cliente quando ele se recupera.
- Cliente broadcast um novo número de época quando se recupera => servidor mata clientes órfãos.
- Requer computação ser completada em T time unidades de tempo. Antigas são simplesmente removidas.

COMUNICAÇÃO CONFIÁVEL EM GRUPO

INTUIÇÃO

A mensagem enviada para um grupo de processos **G** deveria ser entregue para cada membro de **G**. Importante: fazer distinção entre recebimento e entrega de mensagens



COMUNICAÇÃO CONFIÁVEL EM GRUPO (menos simples)

COMUNICAÇÃO CONFIÁVEL NA PRESENÇA DE PROCESSOS COM FALHA

Comunicação em grupo é confiável quando pode ser garantido que a mensagem é recebida e subsequentemente entregue por todos membros sem falha do grupo

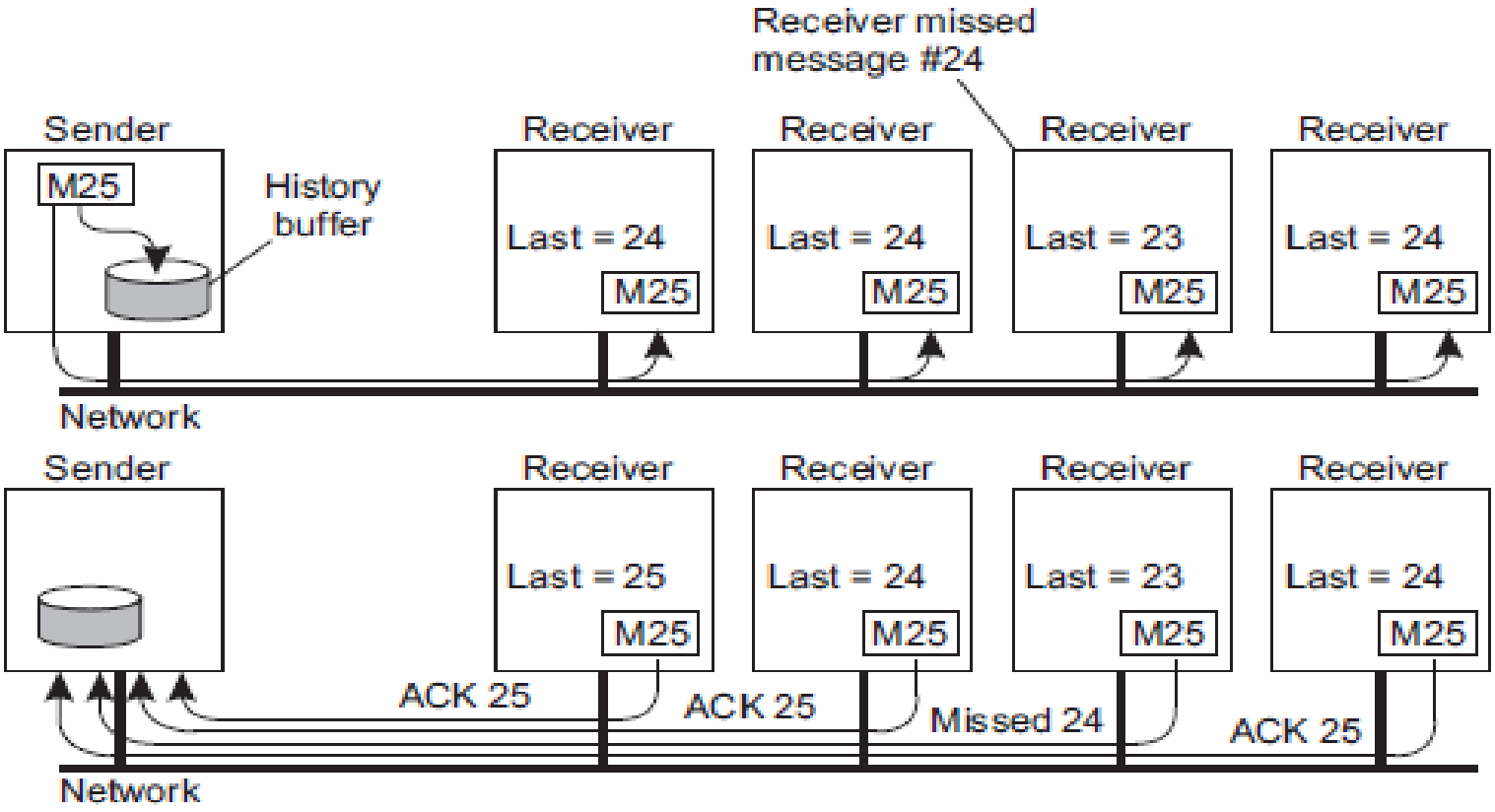
COMUNICAÇÃO CONFIÁVEL NA PRESENÇA DE PROCESSOS COM FALHA

Acordo é necessário sobre como o grupo é antes que uma mensagem recebida possa ser entregue

COMUNICAÇÃO SIMPLES CONFIÁVEL EM GRUPO

COMUNICAÇÃO CONFIÁVEL , MAS ASSUMA PROCESSOS SEM FALHA

Comunicação confiável em grupo agora se torna um multicast confiável: é uma mensagem recebida e entregue para cada destinatário, como pretendido pelo despachante



PROTOCOLOS COMMIT DISTRIBUÍDO

PROBLEMA

Quando uma operação sendo desempenhada por cada membro de um grupo de processos ou nenhum.

- Multicast confiável: uma mensagem é entregue a todos destinatários
- Transação distribuída: cada transação local deve ter sucesso

PROTOCOLOS TWO PHASE COMMIT (2PC)

ESSÊNCIA

O cliente que iniciou a computação age como coordenador; processos requisitados a fazerem commit são os participantes.

- Fase 1a: Coordenador envia VOTE-REQUEST para participantes (também chamado de pre-write)
- Fase 1b: quando o participante recebe VOTE-REQUEST ele retorna VOTE-COMMIT ou VOTE-ABORT para o coordenador. Se ele envia VOTE-ABORT, ele aborta sua computação local
- Fase 2a: Coordenador coleta todos votos; se todos são VOTE-COMMIT, ele envia GLOBAL-COMMIT para todos participantes, ou envia GLOBAL-ABORT
- Fase 2b: Cada participante espera por GLOBAL-COMMIT ou GLOBAL-ABORT e manipula de acordo.

2PC FINITE STATE MACHINE