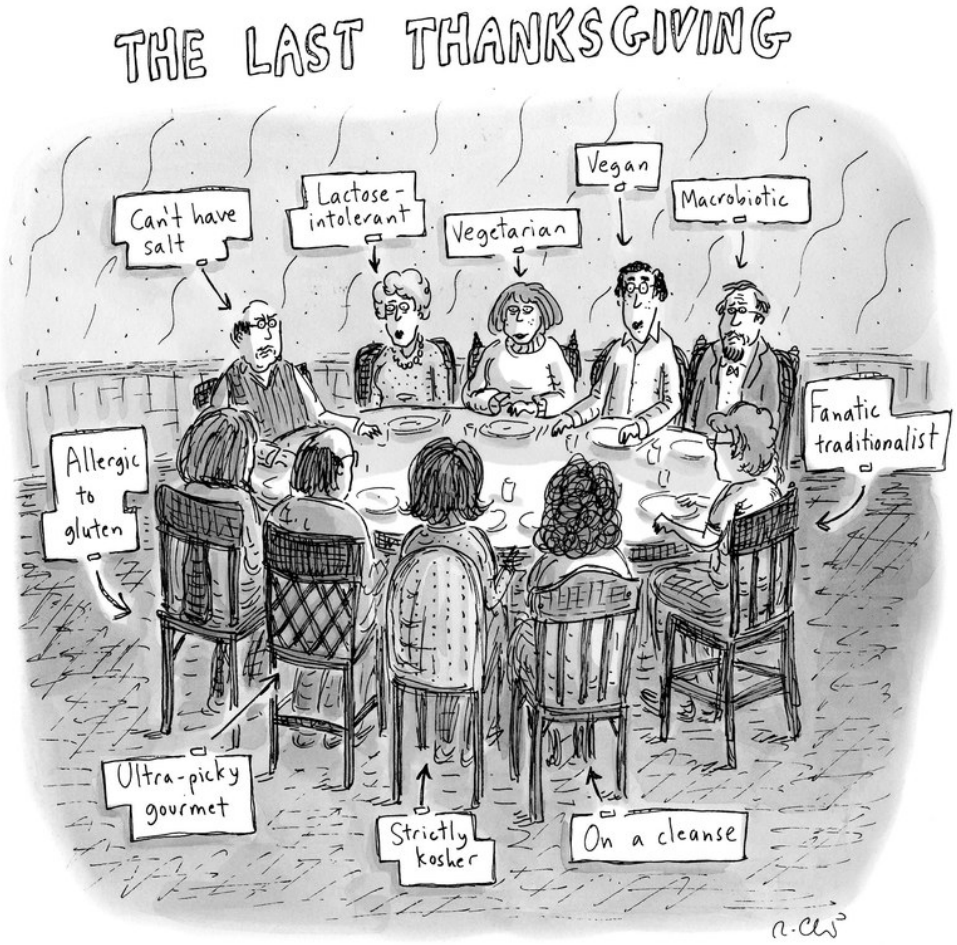
Tolerância a Falhas



Confiança (Dependability)

- Um **componente** disponibiliza **serviços** aos seus **clientes**. Para disponibilizar serviços o componente pode necessitar de serviços de outros componentes → um componente pode **depender** de outro componente.
- Um componente C depende de C' se a correcção de C depender da correcção de C'.

Requisito	Descrição
Disponibilidade	Prontidão para ser utilizado
Fiabilidade	Disponibilização continua de um serviço
Segurança	Baixa probabilidade de catástrofes
Manutenibilidade	Quão fácil é reparar o sistema

Fiabilidade vs Disponibilidade

- Fiabilidade F(t) de um componente C
  - Probabilidade condicional que C esteja a funcionar correctamente durante [0, t[sabendo que C estava a funcionar correctamente em T = 0
- Métricas tradicionais
  - Mean Time To Failure (**MTTF**): Tempo médio até um componente falhar
  - Mean Time To Repair (**MTTR**): Tempo médio necessário para reparar um componente
  - Mean Time Between Failures (**MTBF**): Simplesmente MTTF + MTTR

Fiabilidade vs Disponibilidade

- Disponibilidade D(t) de um componente C
  - Fracção média de tempo em que C esteve a funcionarno intervalo [0, t[
  - Long-term availability A: A(∞)
  - Nota: $\mathrm{A} = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF+MTTR}$
- Falar de Fiabilidade e Disponibilidade só faz sentido quando temos uma noção clara do que é uma falha.

Terminologia

- Fracasso/Falha, Erro, Falha/Culpa

Termo	Descrição	Exemplo
Fracasso/Falha (Failure)	Um componente não está a cumprir com as suas especificações	Programa Crasha
Erro	Parte do programa que pode causar uma falha	Bug
Falha/Culpa (Fault)	Causa do erro	Programador

Terminologia

Termo	Descrição	Exemplo
Prevenção de Falhas	Prevenir a ocorrência de uma falha	Não contratar programadores desleixados
Tolerância a Falhas	Construir um componente tal que possa mascarar a ocorrência de falhas	Cada componente é desenvolvido independentemente por dois programadores
Remoção de Falhas	Reduzir a presença, número, ou seriedade das falhas	Despedir maus programadores
Previsão de falhas	Estimar a presença actual, incidência futura e consequências de falhas	Estimar como se estão a portar os RH na contratação de maus programadores

(Falhas = Fault)

Modelos de Fracasso (Failure)

Tipos de fracassos	Descrição do comportamento do servidor
Crash fracassado	Para, mas continua a funcionar até parar
Omissões - Omissão na recepção - Omissão no envio	Não responde a mensagens recebidas Não recebe novas mensagens Não envia mensagens
Falha temporal	Responde fora dos limites temporais especificados.
Falha na resposta - Falha no valor - Falha transição de estado	Resposta é incorrecta O valor da resposta está errado Desvio do controlo de fluxo correcto
Falha Arbitrária	Pode produzir respostas arbitrária em tempos arbitrários

Confiança vs Segurança

- Omissão vs Comissão
  - Uma falha arbitrária pode ser classificada como maliciosa.
  - **Falhas por omissão**: um componente falha a tomada de uma acção que devia ter tomado
  - **Falhas por comissão**: um componente toma uma acção que não devia ter tomado
- ATENÇÃO: uma falha deliberada é tipicamente um problema de segurança. Distinguir entre umas e outras é no entanto geralmente impossível.

Falhas terminais (Halting failures)

- Cenario:
  - C não percepciona qualquer actividade de C' – Falha terminal ?
    - Distinguir entre um crash ou uma falha por omissão/temporal pode ser impossível
- Sistemas Assíncronos vs Síncronos
  - Sistema assíncrono: não se assume nada sobre o velocidade de execução de um processos ou tempos de entrega de mensagens → **não se pode detectar falhas por crash de forma fiável.**
  - Sistema síncrono: tempos de execução e entrega de mensagens estão delimitados → **podemos detectar de forma fiável falhas por omissão e e temporais**
  - Na prática temos sistemas parcialmente síncronos: maior parte das vezes podemos assumir que o sistema se comporta sincronamente, apesar de não existirem limites aos momentos em que se comporta de forma assíncrona → **podem ser detectadas falhas por crash normalmente**

Falhas terminais

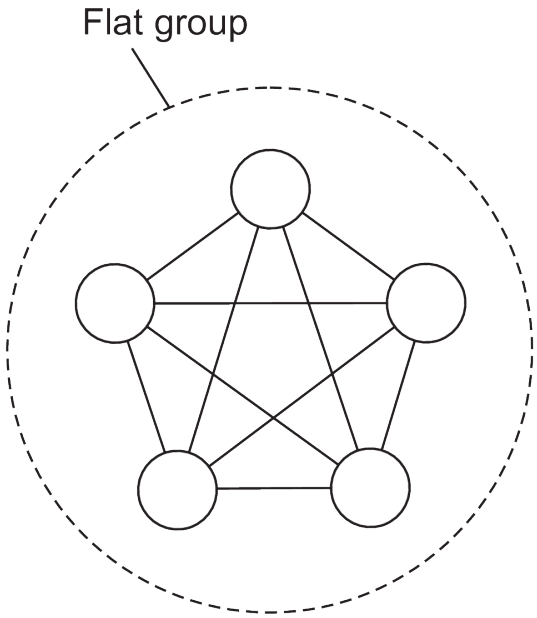
Tipo de paragem	Descrição
Fail-stop	Falha por crash, mas detectada fiavelmente
Fail-noisy	Falha por crash, eventualmente detectavel
Fail-silent	Falhas por omissão ou crash: cliente não distingue o que aconteceu
Fail-safe	Arbitrária, mas benigna (i.e. não fazem estrago)
Fail-arbitrary	Arbitrária, com falhas maliciosas

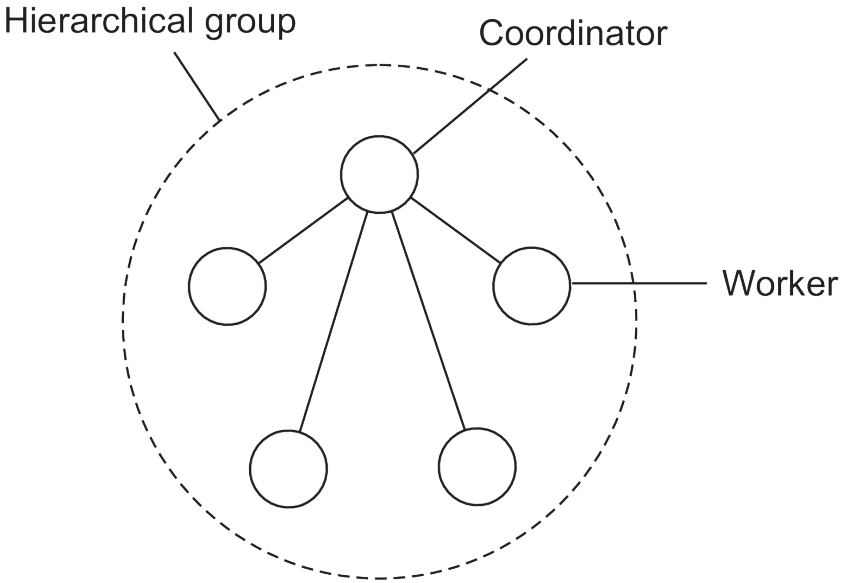
Redundancia para mascarar falhas

- Tipos de redundância
  - Informação: Acrescentar bits extra às unidades de dados para que os erros possam ser recuperados quando os bits são danificados
  - Temporal: Desenhar o sistema de tal forma que uma acção possa ser desempenhada novamente caso algo de errado aconteça. Tipicamente usado quando as falhas são transientes ou intermitentes.
  - Fisica: Acrescentar equipamento ou processos por forma a permitir que um ou mais componentes possam falhar. Este tipo é usado extensivamente em sistemas distribuídos.

Resiliência Processo

- Proteger contra o mau funcionamento de um processo através da replicação de processos, organizando múltiplos processos em grupos de processos. Distinguir entre **grupos planos** e **grupos hierárquicos**





Grupos e mascáras de falha

- K-fault tolerant group
  - Quando um grupo consegue mascarar quaisquer falhas de membros concorrencialmente (k é chamado de grau de tolerância a falhas)
- Quão grande precisa de ser um K-fault tolerant group ?
  - Com falhas terminais (crash/omissão/temporais): precisamos de k+1 membros, nenhum membro irá produzir um resultado incorrecto pelo que um único membro é suficiente.
  - Com falhas arbitrárias: necessário 2k+1 membros para que o resultado correto possa ser obtido por maioria de votos.
- Importantes assunções:
  - Todos membros são iguais
  - Todos membros processam os comandos pela mesma ordem
- Temos a certeza que todos processos fazem o mesmo

Consenso

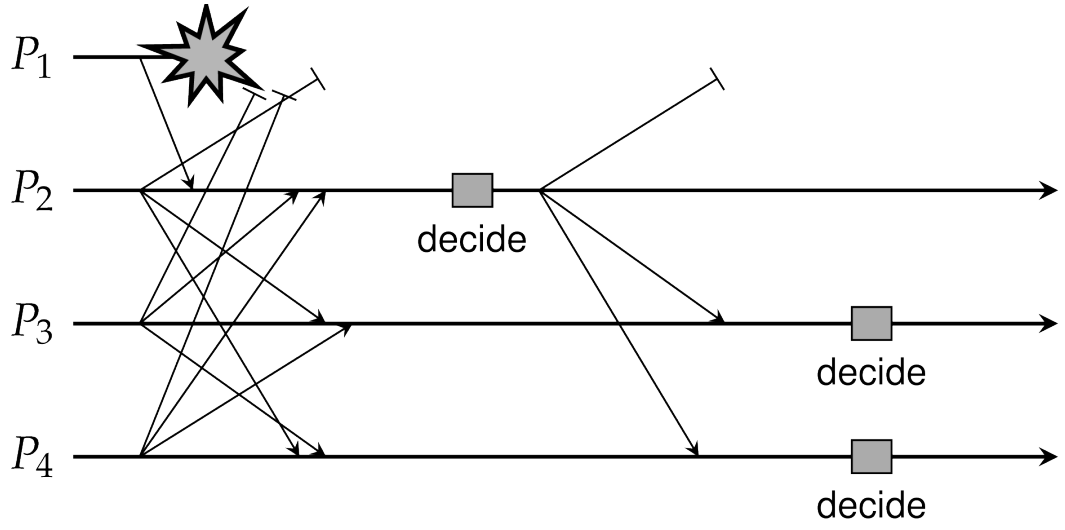
- Pré-requisito:
  - Num grupo de processos tolerantes a falhas, cada processo que não esteja a falhar executa os mesmos comandos, e pela mesma ordem, que qualquer outro processo não em falha.
- Reformulação
  - Membros do grupo que não falham precisam de chegar a um consenso sobre qual comando executar de seguida

Consenso por flooding

- Modelo de Sistema
  - Um grupo de processos $P = \{P_1, \ldots, P_2\}$
  - Semântica de falha Fail-stop, isto é com detecção fiável de falhas
  - Um cliente contacta P_i pedindo para executar um comando
  - Cada P_i mantém uma lista dos comandos propostos.
- Algoritmo (com base em rondas)
  - Na ronda r, P_i envia por multicast o seu conjunto C_i^r para todos os outros
  - No final de r, cada P_i funde todos os comandos recebidos num novo C_i^{r+1}
  - Próximo comando Cmd_i é selecionado por uma função determinística partilhada globalmente

Consenso por flooding: Exemplo

- P_2 recebe todos comandos proposto por todos outros processos → toma decisão
- P_3 pode detectar que P_1 crashou, mas não sabe se P_2 recebeu alguma coisa, i.e, P_3 não tem como saber se tem a mesma informação que P_2 → não pode tomar decisão (mesmo para P_4)



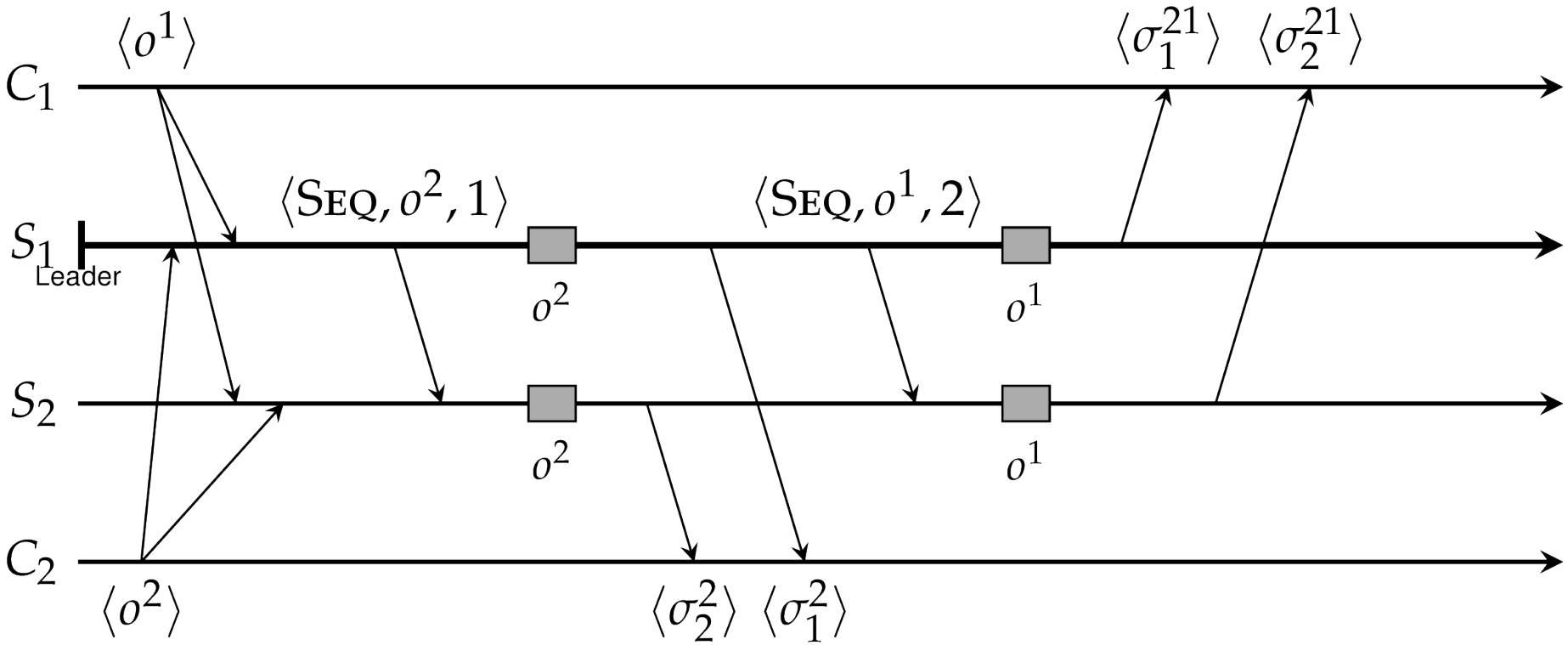
Consenso Realista: Paxos

- Assunções (fracas, e realistas)
  - Sistema é **parcialmente síncrono** (na realidade até pode ser assíncrono)
  - **Comunicação** entre processos pode ser **instável**: mensagens pode ser perdidas, duplicadas ou reordenadas
  - **Mensagens corrompidas podem ser detectadas** (e subsequentemente ignoradas)
  - Todas **operações são determinísticas**: uma vez iniciada uma execução, é sabido o que irá fazer.
  - Processos podem exibir **falhas por crash**, mas **não falhas arbitrárias**.
  - Processos **não podem conspirar.**

Principios Paxos

- Inicio
  - É assumida uma configuração cliente-servidor, inicialmente com um **servidor primário**
  - Para tornar o servidor mais robusto, começamos por adicionar um **servidor de backup.**
  - Para garantir que todos os comandos são executados pela mesma ordem em ambos os servidores, o primário atribui **números de sequencia únicos** a todos os comandos. No Paxos o primário é chamado de **líder**.
  - Assumir que o comandos actuais podem sempre ser restaurados (tanto através dos clientes ou servidores) → apenas consideradas mensagens de controlo.

Situação com dois servidores

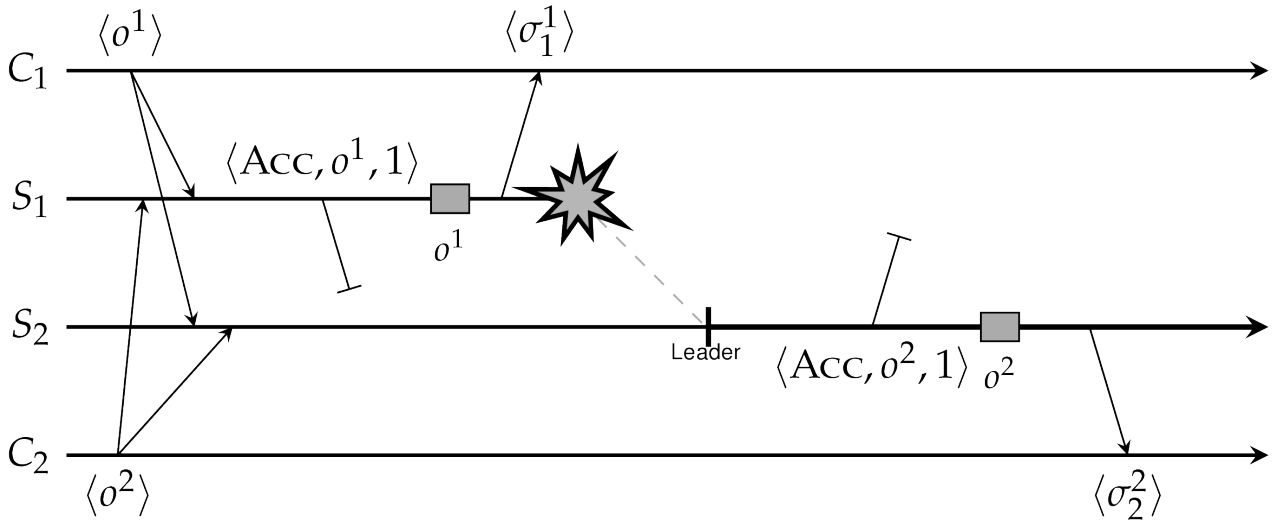


Tratar mensagens perdidas

- Alguma terminologia Paxos
  - O líder envia mensagem de aceitação ACCEPT(o, t) para os backups quando atribui um timestamp t ao comando o.
  - O backup responde enviando uma mensagem de aprendizagem: LEARN(o, t)
  - Quando o líder repara que a operação o não foi ainda aprendida, retransmite ACCEPT(o, t) com o timestamp original.

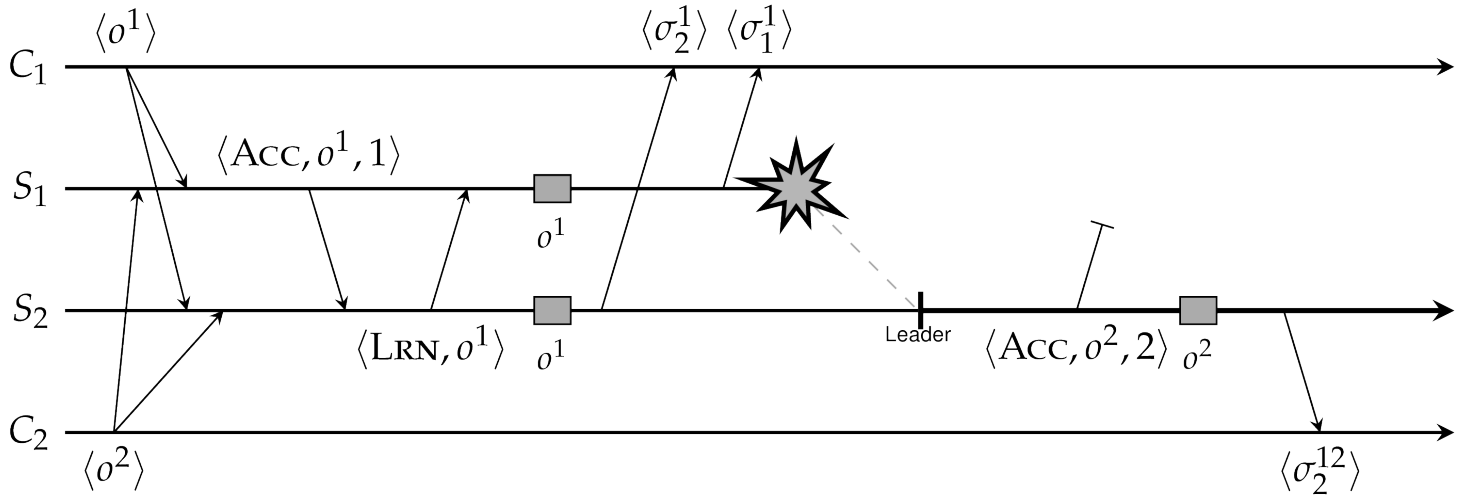
Dois servidores e um crash: problema

- Primario crash após executar uma operação, mas o backup nunca recebeu a mensagem de aceitação.



Dois servidores e um crash: solução

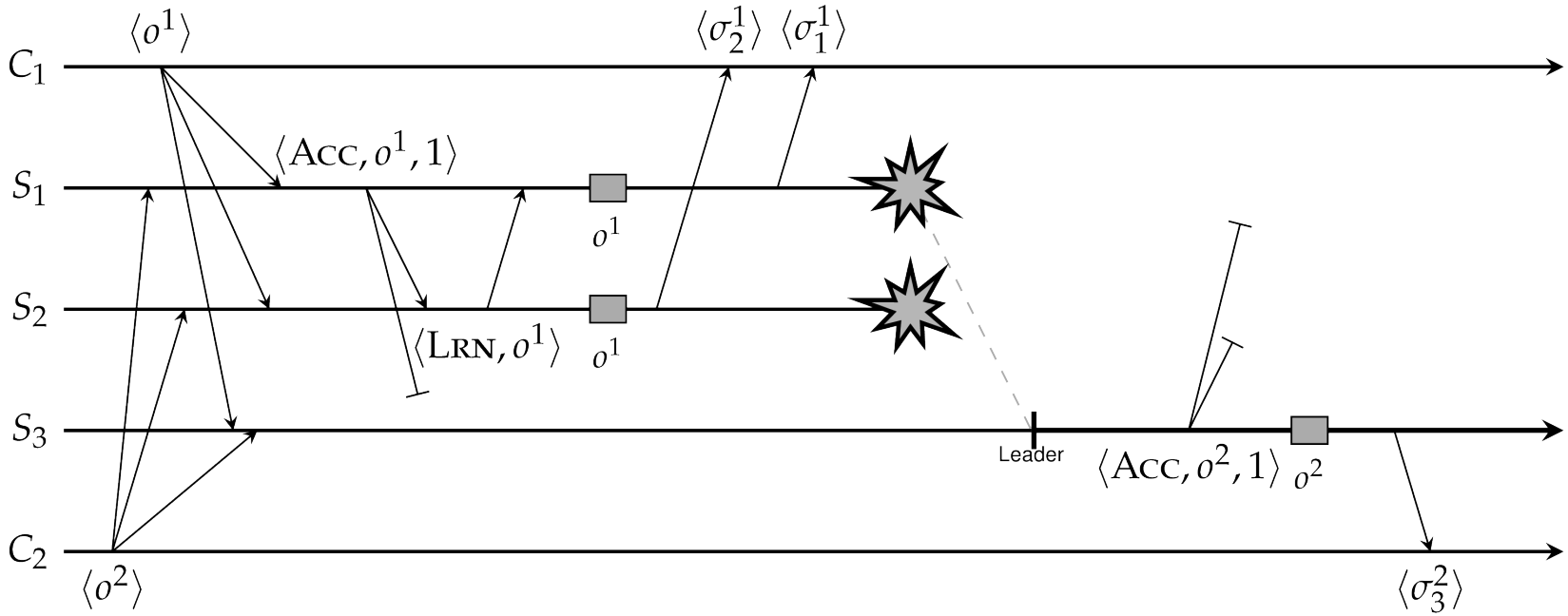
- Solução:
  - Nunca executar uma operação antes de a mesma ter sido aprendida.



Três servidores e dois crashes: problema?

Que acontece quando LEARN(o^1) é enviado de S_2 para S_1 e é perdido?

S_2 necessita esperar que S_3 tenha aprendido o^1

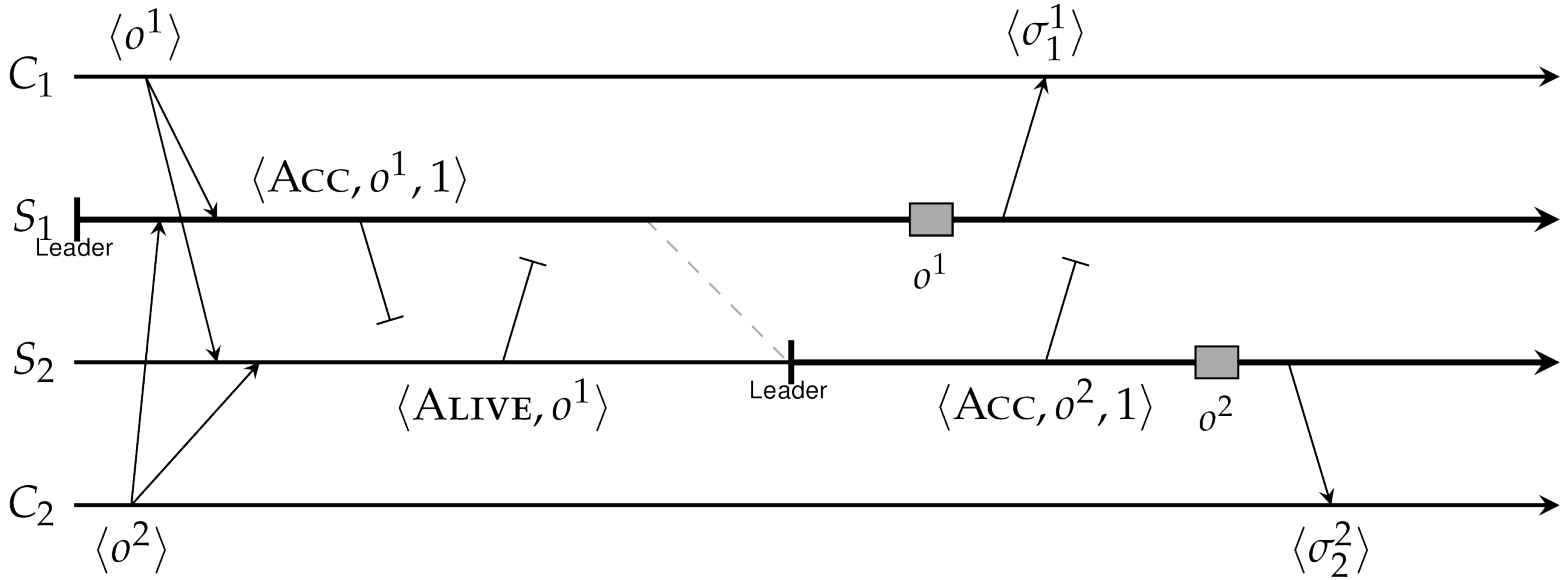


Paxos: Regra fundamental

- No Paxos, um servidor S não pode executar uma operação o sem que tenha recebido LEARN(o) de todos os outros servidores operacionais

Detecção de falhas

- Detecção fiavel de falhas é praticamente impossivel. A solução passa por um conjunto de timeouts, mas é necessario ponderar que a detecção de uma falha é muitas vezes um falso positivo.



Numero de Servidores necessários

- Pelo menos **três** (3) servidores
- No Paxos com 3 servidores, o servidor S não pode executar uma operação o sem que tenha recebido pelo menos uma (outra) mensagem LEARN(o), de maneira a saber que uma maioria de servidores irá executar o.
- Se um dos servidores de backup crashar, Paxos continua a funcionar corretamente: operações nos servidores operacionais ocorrem pela mesma ordem.

Leader crasha depois de executar o^1

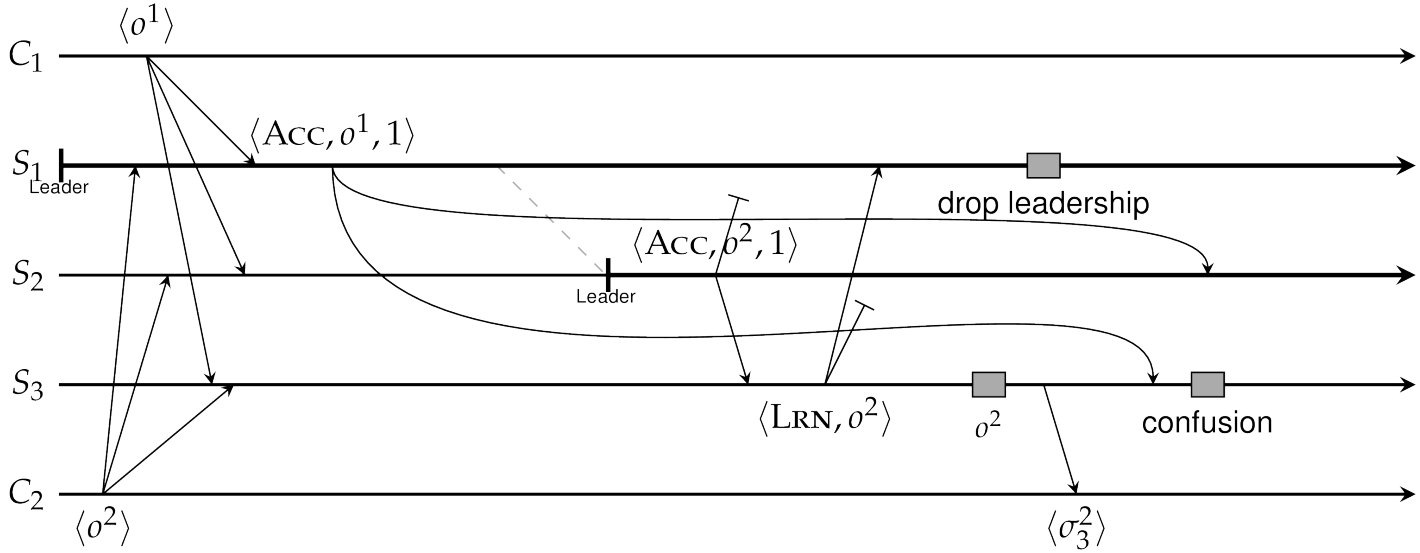
- S_3 ignora completamente a actividade de S_1
  - S_2 recebeu ACCEPT(o, 1), detecta crash, torna-se no leader
  - S_3 nem sequer recebeu ACCEPT(o, 1)
  - S_2 envia ACCEPT(o^2, 2) → S_3 vê um timestamp inesperado e informa S_2 que perdeu o^1
  - S_2 retransmite ACCEPT(o^1, 1), o que permite a S_3 recuperar
- S_2 não recebe ACCEPT(o^1, 1)
  - S_2 detectou o crash e torna no leader
  - S_2 envia ACCEPT(o^1, 1) → S_3 retransmite LEARN(o^1)
  - S_2 envia ACCEPT(o^2, 2) → S_3 informa S_2 que aparentemente perdeu ACCEPT(o^1, 1) de S_1 pelo que S_2 pode recuperar

Leader crasha depois de enviar ACCEPT(o^1, 1)

- S_3 ignora completamente a actividade de S_1
  - Assim que S_2 anuncia que o^2 é para ser aceite, S_3 vai reparar que perdeu uma operação e que pode pedir a S_2 ajuda para a recuperar
- S_2 não recebe ACCEPT(o^1, 1)
  - Assim que S_2 propõe uma operação, vai utilizar um timestamp obsoleto, o que permite a S_3 informar S_2 que perdeu a operação o^1
- Paxos (com três servidores) comporta-se correctamente quando um servidor crasha, independentemente de quando o crash ocorreu.

Detecção de falsos crashes

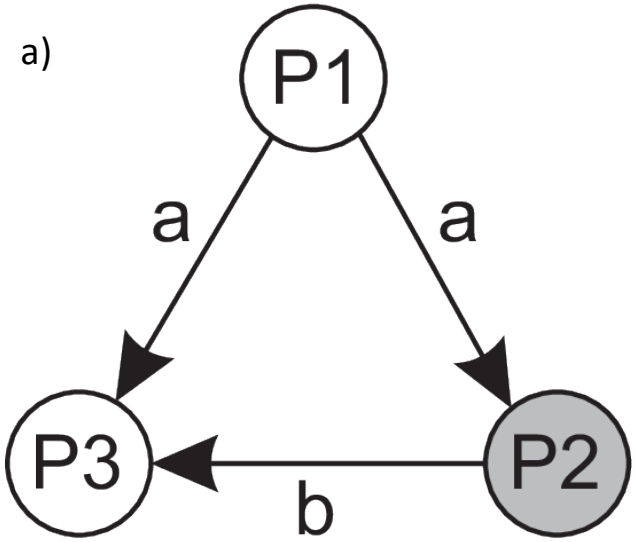
- S_3 recebe um ACCEPT(o^1, 1), mas muito depois de ACCEPT(o^2, 1). Se soubesse quem é o leader actual, poderia rejeitar a mensagem atrasada.
  - **Leaders devem incluir o seu ID nas mensagens**



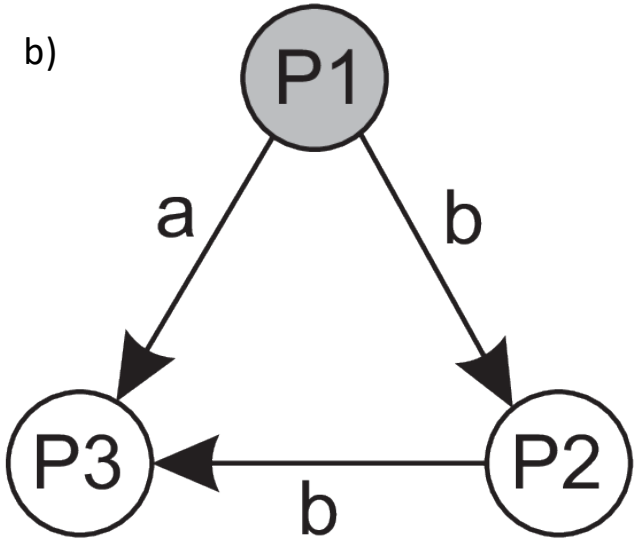
Consenso em falhas arbitrárias

- Considerando grupos de processo em que a comunicação entre processos é inconsistente: (a) mau encaminhamento de mensagens ou (b) comunicar coisas diferentes a diferentes processos.

a)



b)

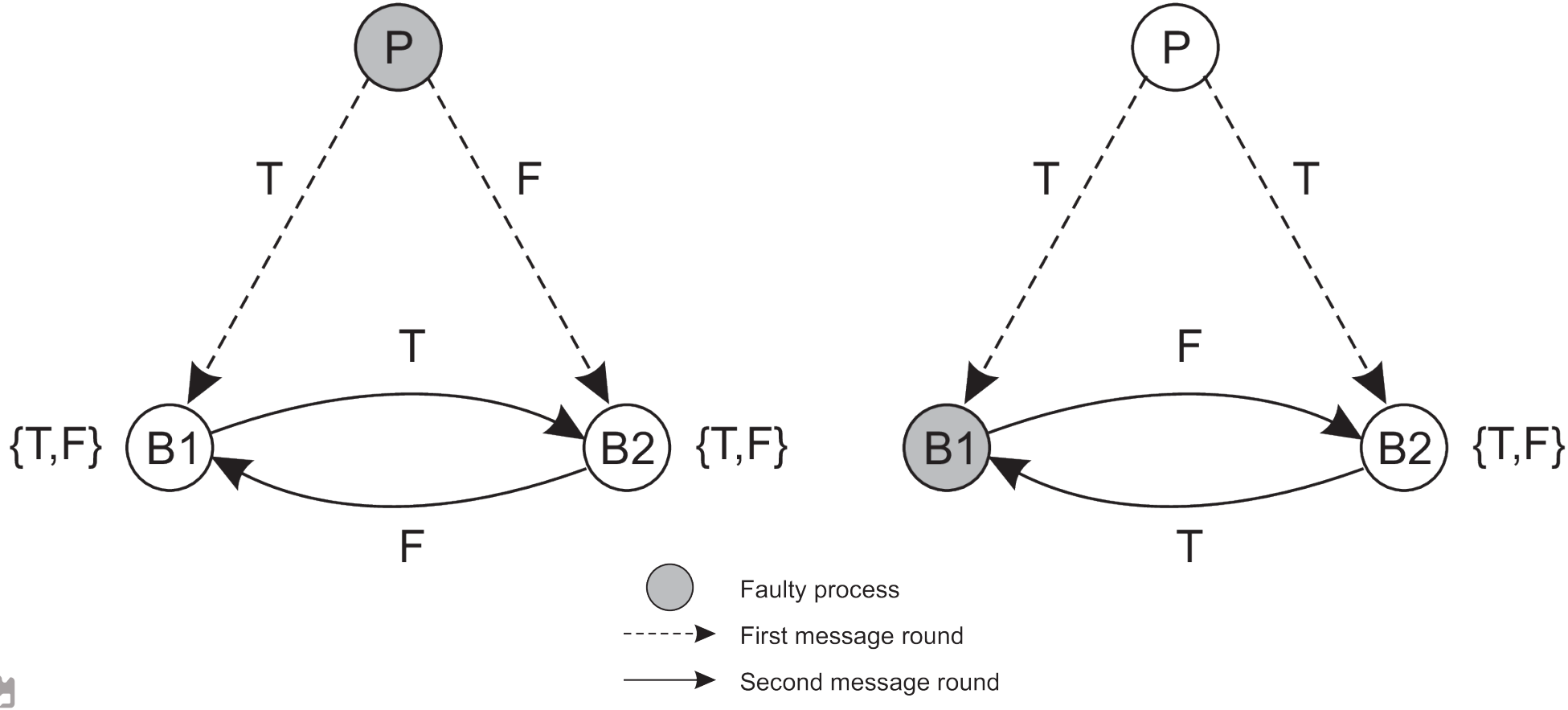


a)

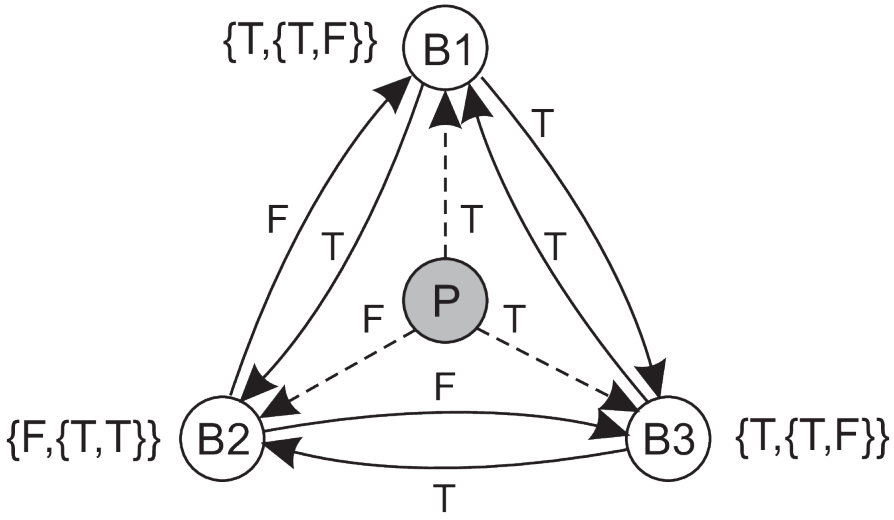
Consenso em falhas arbitrárias

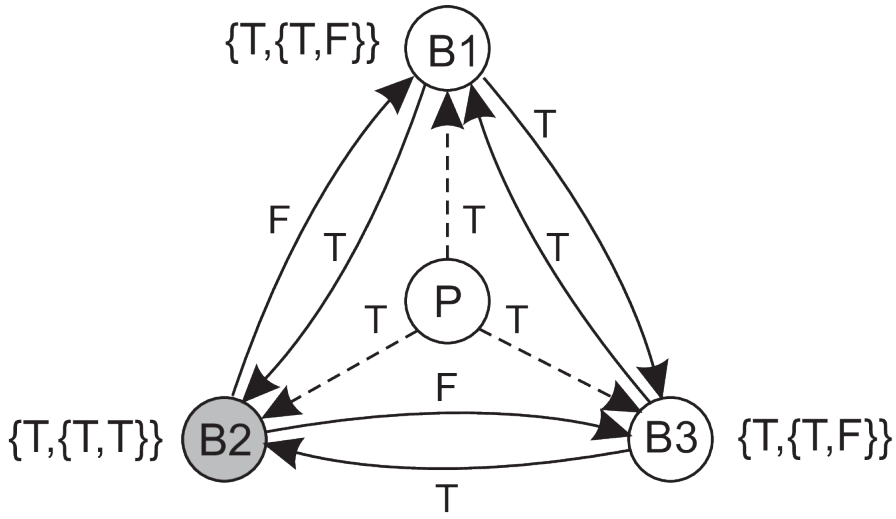
- Modelo Sistema
  - Consideramos um primário P e n-1 backups B_1, ..., B_{n-1}
  - Um cliente envia $v \in \{T, F\}$ a P
  - Mensagens podem se perder, mas tais são detectadas
  - Mensagens não podem ser corrompidas sem haver detecção
  - O receptor de uma mensagem detecta fielmente que foi o emissor
- Requisitos para um Acordo Bizantino (BA)
  - BA1: Todos os backups que não estejam em falha armazenam o mesmo valor
  - BA2: Se o primário não estiver a falhar então todos os backups que não estejam a falhar armazenam exactamente o que o primário enviar.

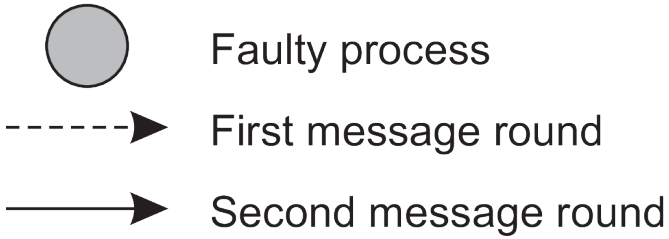
Porque ter 3k processos não é suficiente



Porque ter 3k+1 processos é suficiente



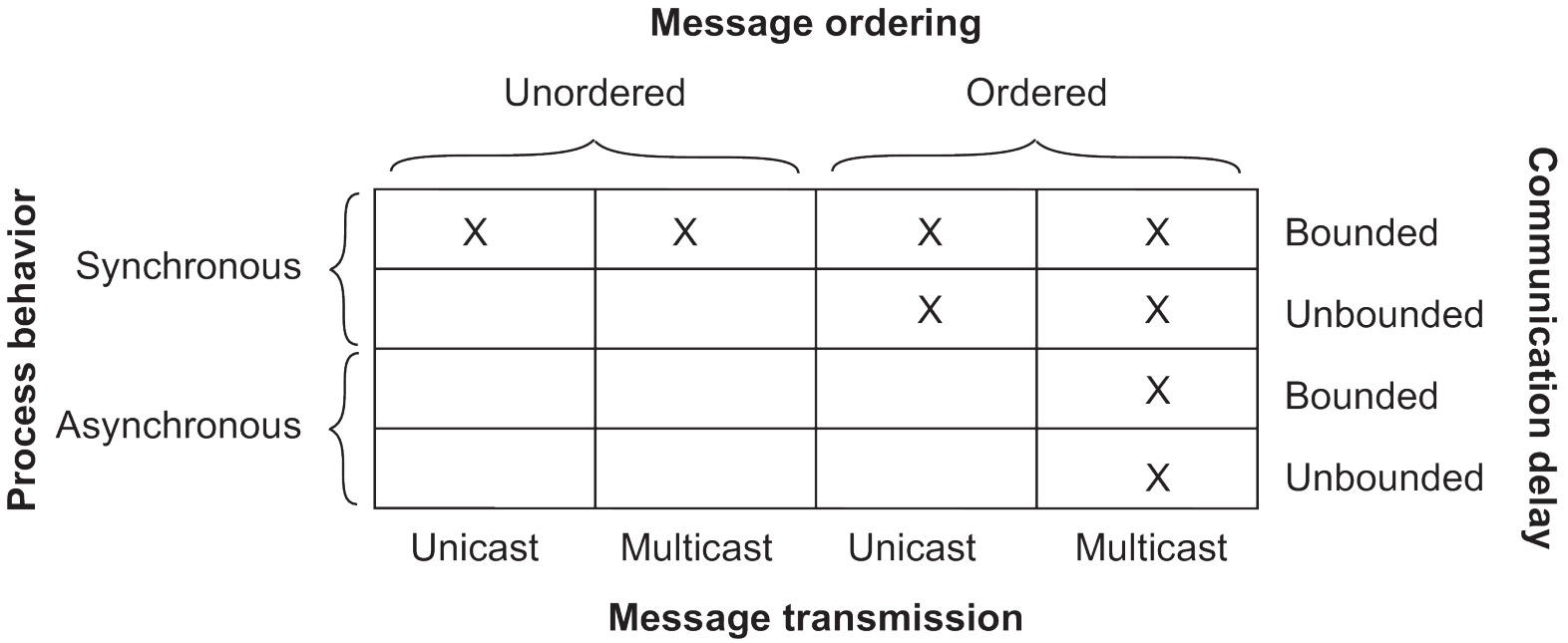




Concretizar tolerância a falhas

- Considerando que os membros de um grupo tolerante a falhas encontram-se fortemente acoplados, podemo-nos deparar com impactos significativos na performance, mas até mesmo em situações em que é impossível concretizar tolerância a falhas.
- Quais são as limitações ao que se pode conseguir ?
  - O que é necessário para permitir alcançar um consenso ?
  - O que acontece quando um grupo se encontra particionado ?

Consenso distribuído: Quando é alcançável ?



- Requisitos formais para consenso:
  - Processos produzem o mesmo valor
  - Todos resultados são válidos
  - Todos processos têm que eventualmente fornecer um resultado

Consistência, disponibilidade e particionamento

- Teorama CAP
  - Qualquer sistema distribuído que partilhe informação só pode ter duas das seguintes três propriedades:
  - **C**onsistência, qualquer informação partilhada e replicada aparece com um único valor actualizado
  - **A**vailability (Disponibilidade), que garante que uma actualização será sempre eventualmente executada
  - **P**artição do grupo será sempre tolerada
- Corolário
  - Numa rede com falhas de comunicação é impossível realizar uma operação atómica de leitura/escrita em **memória partilhada** que garanta uma resposta a todos pedidos.

Detecção de falhas

- Como podemos detectar fielmente que um processo crashou ?
- Modelo geral
  - Cada processo está equipado com um modulo de detecção de falhas
  - Um processo P sonda outro processo Q por uma reacção
  - Se Q reagir: Q é considerado disponível (por P)
  - Se Q não reagir no intervalo de tempo t: Q fica sob suspeição de ter crashado
- Num sistema **síncrono**:
  - Uma suspeita de crash é na realidade uma certeza

Detecção prática de falhas

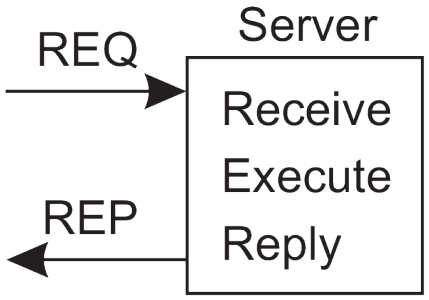
- Se P não receber um heartbeat de Q dentro do intervalor t: P suspeita de Q
- Se Q mais tarde enviar uma mensagem (que é recebida por P):
  - P deixa de suspeitar de Q
  - P aumenta o valor do intervalor t
- Atenção: se Q tiver crashado, P irá continuar a suspeitar de Q.

RPC’s fiáveis

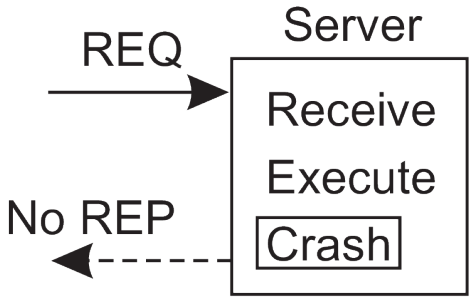
- O que pode acontecer de errado?
  - Cliente não encontra o servidor
  - A mensagem com o pedido do cliente para o servidor é perdida
  - O servidor crash depois de receber o pedido
  - A resposta do servidor para o cliente é perdida
  - O cliente crasha depois de enviar o pedido
- Soluções:
  - (falha na localização): reportar ao cliente
  - (pedido perdido): voltar a enviar

RPC's fiáveis: servidor crasha

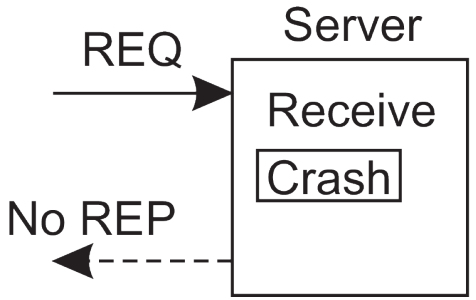
- Problema
  - Enquando (a) é o caso normal, situações (b) e (c) necessitam de soluções diferentes. Apesar de tudo não sabemos o que aconteceu.
- Duas aproximações:
  - At-least-once-semantics: O servidor garante que irá executar a operação pelo menos uma vez, de qualquer maneira
  - At-most-once-semantics: O servidor garante que irá executar a operação no máximo uma vez.



(a)



(b)



(c)

Porque é impossível recuperar de uma falha no servidor transparentemente ?

- Tres eventos distintos no servidor
  - M: envio de uma mensagem completa
  - P: completa o processamento de um documento
  - C: crash
- Seis ordens diferentes
  - M→P →C: Crash depois de reportar termino
  - M→C →P: Crash depois de reportar o termino, mas antes de actualizar
  - P →M →C: Crash depois de reportar o termino, e depois de actualizar
  - P →C (→M): Actualização teve lugar, e depois crashou
  - C (→P →M): Crash antes de fazer qualquer coisa
  - C (→M →P): Crash antes de fazer qualquer coisa

Porque é impossível recuperar de uma falha no servidor transparentemente ?

Reissue strategy	Strategy M → P			Strategy P → M		
	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**					

OK=Document processed once

DUP=Document processed twice

ZERO=Document not processed at all

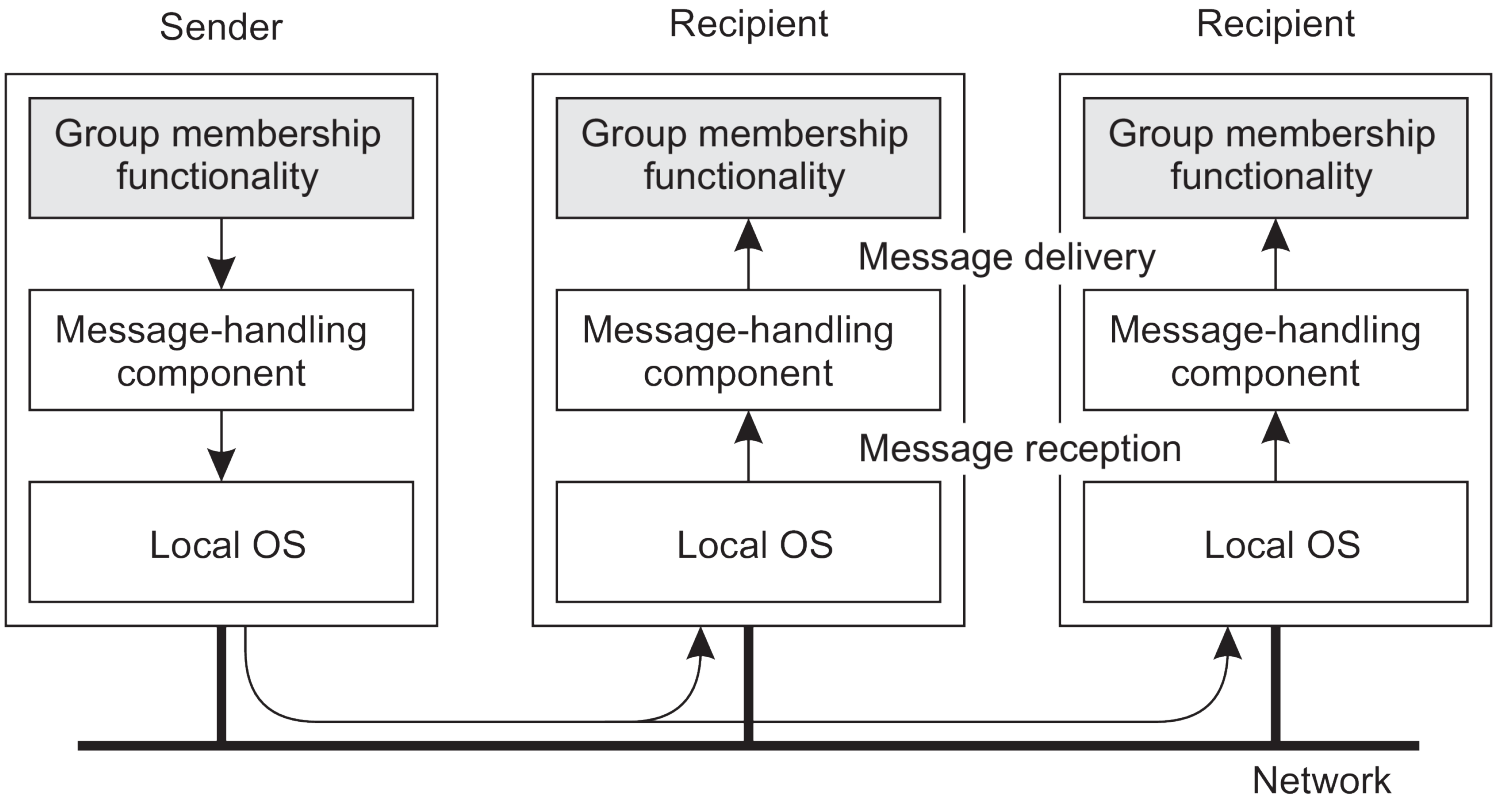
RPC fiável: perda de mensagens

- O que o cliente repara, é que não está a receber uma resposta. No entanto, não tem como saber se a causa é perda do pedido, servidor crashou ou perdeu a resposta.
- Solução (parcial)
  - Desenhar o servidor de maneira que as operações sejam idempotentes: repetir uma operação é o mesmo que realizar apenas uma vez:
    - Operações de leitura puras
    - Operações de substituição restritas
  - Muitas operação são idempotentes por natureza, tais como as transacções bancárias.

RPC fiável: cliente crasha

- Problema
  - Servidor está fazer trabalho e ocupar recursos para nada (computação órfã)
- Solução
  - Orfão é morto (ou é feito um retrocesso) pelo cliente quando este recupera
  - Cliente envia um broadcast com um novo numero de época quando recupera → servidor mata os órfãos do cliente
  - Requerer que uma computação termine no máximo em T unidades de tempo. Antigas são simplesmente removidas

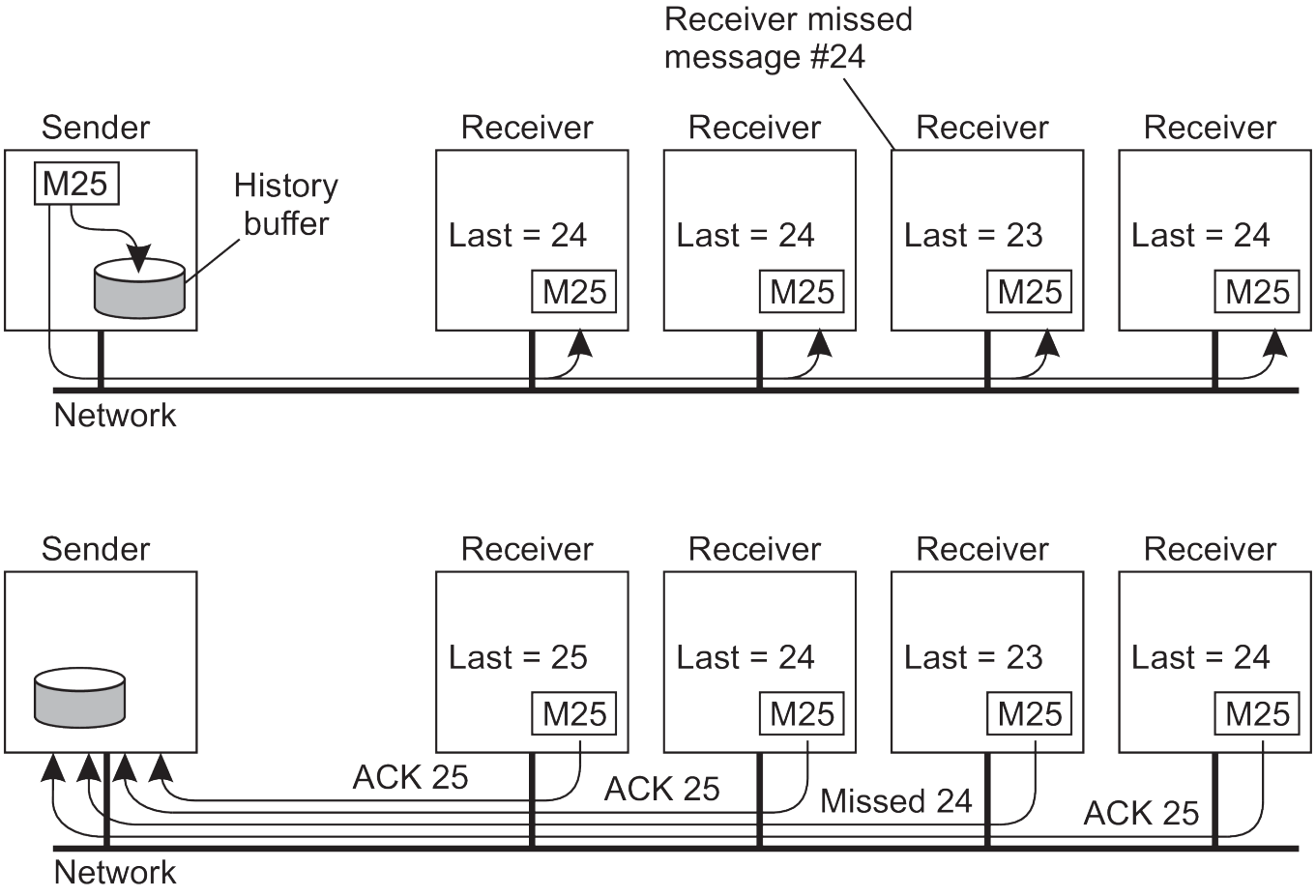
Comunicação em grupo fiável e simples



Comunicação em grupo fiável (não simples)

- Comunicação fiável na presença de processos em falha
  - A comunicação é fiável quando se pode garantir que a mensagem é recebida e subsequentemente entregue por todos os membro do grupo não em falha.
- Dificuldade
  - Um acordo é necessário sobre qual é o grupo antes da mensagem ser entregue.

Comunicação em grupo fiável e simples



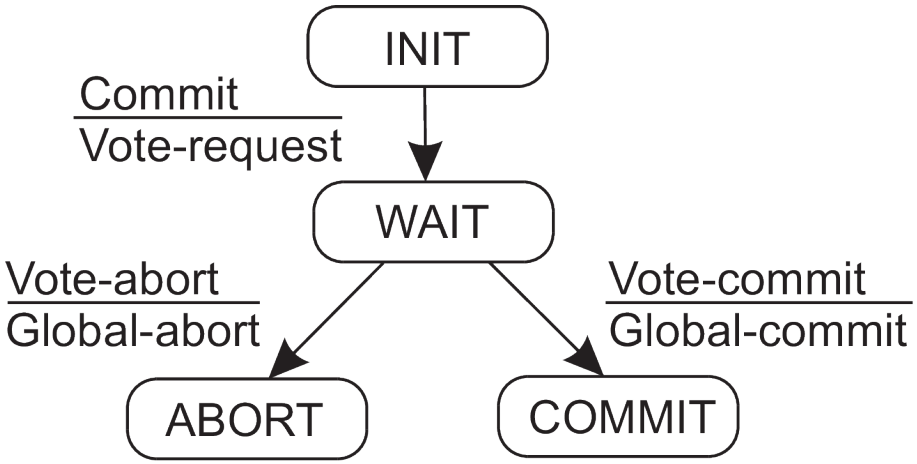
Protocolos de commit distribuído

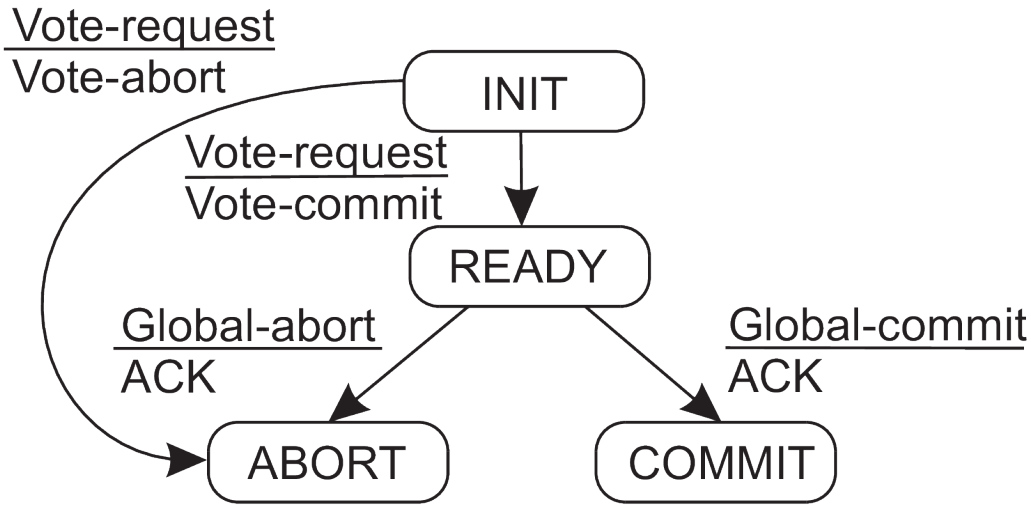
- Problema
  - Garantir que uma operação é executada por todos membros de um grupo, ou que é executada por nenhum (tudo ou nada)
    - Multicast fiável: uma mensagem é entregue a todos os receptores
    - Transacção distribuída: cada transacção local tem que ter sucesso

Protocolo de commit em 2-passos (2PC)

- O cliente que inicia a computação age como **coordenador**; processos que necessitam tomar parte do commit são **participantes**.
  - Fase 1a: Coordenador envia VOTE-REQUEST aos participantes (também chamado de **pre-write**)
  - Fase 1b: Quando participante recebe VOTE-REQUEST responde com VOTE-COMMIT ou VOTE-ABORT para o coordenador. Se enviar um VOTE-ABORT, aborta a sua computação local
  - Fase 2a: Coordenador coleciona todos votos; se todos forem VOTE-COMMIT, envia um GLOBAL-COMMIT a todos participantes, caso contrário envia um GLOBAL-ABORT
  - Fase 2b: Cada participante espera por GLOBAL-COMMIT ou GLOBAL-ABORT e processa de acordo.

2PC – Máquinas de estados finitos





2PC – Participante falha

- Participante crasha no estado S, e recupera para S
  - INIT: Não há problema, participante não conhecia o procotolo
  - READY: Participante está à espera de commit ou abort. Após recuperar, participante precisa de saber que transição de estado precisa efectuar → guarda a decisão do coordenador
  - ABORT: Simplesmente tornar o estado de abort idempotente, isto é, remover os resultados do workspace
  - COMMIT: Também tornar o estado de commit idempotente, isto é, copiar o workspace para storage
- Quando um commit distribuído é necessário, ter participantes que usam um workspace temporário para guardar resultado permite uma recuperação na presença de falhas.

2PC – Coordenador Falha

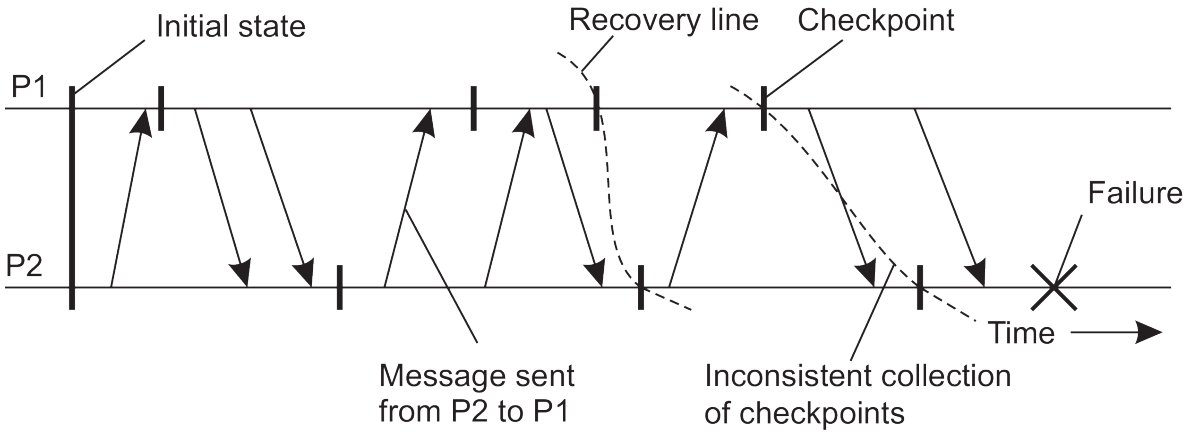
- O principal problema é o facto de que a decisão final do coordenador pode não estar disponível (ou até mesmo ser perdida)
- Para um participante P no estado READY, definir um tempo máximo t para esperar uma decisão do coordenador. P deverá descobrir o que os outros participantes conhecem.
- Um participante não pode decidir localmente, depende sempre dos outros processos (eventualmente em falha)

Recuperação de Falhas

- Quando uma falha acontece, é necessário trazer o processo devolta a um estado sem falhas:
  - Avançar no erro (Forward error recovery): Encontrar um novo estado em que o Sistema possa continuar a operar
  - Retroceder no erro (Backward error recovery): Voltar **atrás** no Sistema a um estado livre de erros
- Backward error recovery é o mais usual, mas necessita do estabelecimento de pontos de recuperação (**recovery points**)
- A recuperação em sistemas distribuidos é complicada pelo facto dos processos terem necessidade de cooperar na identificação de um **estado consistente**, apartir do qual possam recuperar

Recuperação de estado consistente

- Requisito
  - Para cada mensagem que tenha sido recebida é demonstravél que a mesma foi enviada.
- Linha de recuperação
  - Assumindo processos que fazem checkpoint regularmente, é a mais recente checkpoint

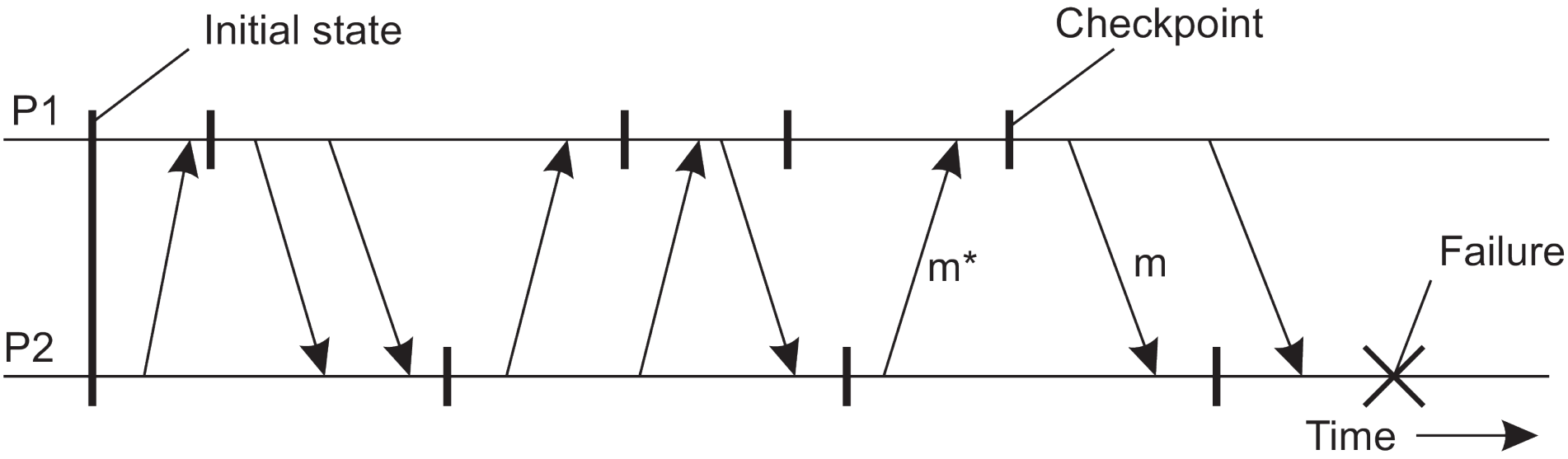


Checkpoint coordenado

- Cada processo faz um checkpoint após uma acção coordenada globalmente
- Solução Simples:
  - Usar um protocolo bloqueante em duas fases
    - O coordenador envia em multicast a mensagem de pedido de checkpoint (checkpoint request)
    - Quando um participante recebe tal mensagem, faz um checkpoint, para o envio de mensagens (app), e reporta de volta que fez um checkpoint
    - Quando todos checkpoints forem confirmados pelo coordenador, este anuncia por broadcast que o checkpoint foi feito (checkpoint done)
  - É possível nos cingirmos apenas aos processos que dependem da recuperação do coordenador e ignorar os demais.

Reversão em Cascata

- Se um checkpoint é feito nos instantes “errados”, a linha de recuperação pode localizar-se no inicio de todo sistema, a esta situação dá-se o nome “Reversão em Cascata”



Checkpoints independentes

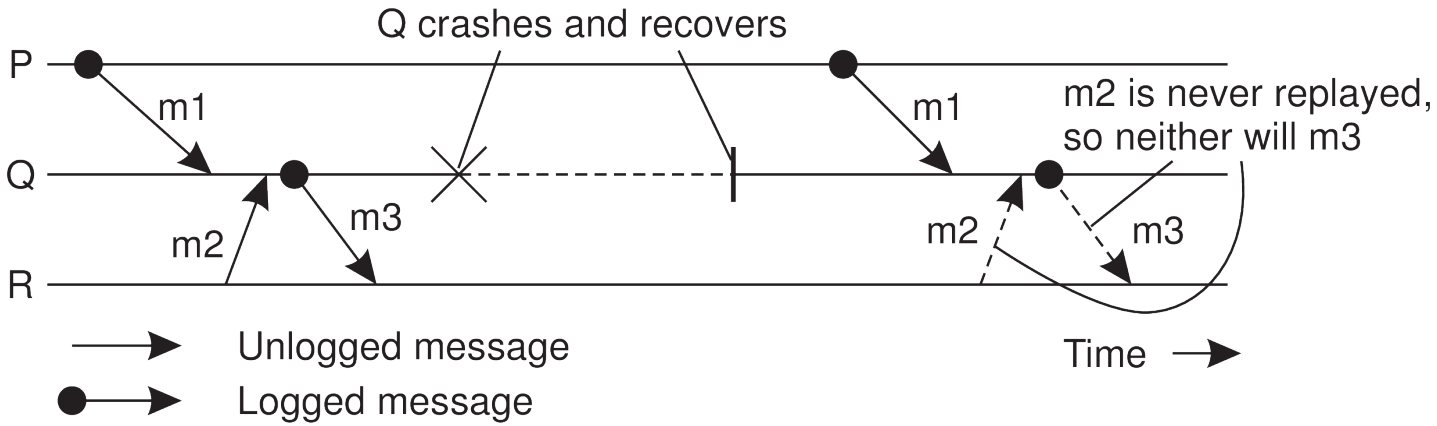
- Cada processo cria checkpoints de forma independente, com o risco de uma reversão em cascata.
  - Seja $CP_i(m)$ o $m^{\underline{o}}$ checkpoint do processo P_i e $INT_i(m)$ o intervalo entre $CP_i(m-1)$ e $CP_i(m)$
  - Quando o processo P_i envia uma mensagem no intervalo $INT_i(m)$, a mesma faz piggyback (i, m)
  - Quando o processo P_j recebe a mensagem no intervalo $INT_j(n)$ guarda a dependência $INT_i(m) \rightarrow INT_j(n)$
  - A dependência $INT_i(m) \rightarrow INT_j(n)$ é guardada junto com o checkpoint $CP_j(n)$
- Se o processo P_i retroceder para $CP_i(m-1)$, P_j tem que retroceder para $CP_j(n-1)$

Registo de mensagens (logging)

- Em vez de assumir um checkpoint (caro), tentar repetir as mensagens e comportamentos desde o ultimo checkpoint → guardar registo de mensagens.
- Assume-se um modelo de execução à peça e determinístico
  - A execução de cada processo pode ser considerada uma sequencia de estados intervalados.
  - Cada intervalo entre estados começa com um evento não determinístico (ex. recepção de uma mensagem)
  - Execução no intervalo é determinística
- Se registarmos os eventos não determinísticos (para repetir mais tarde), obtemos um modelo de execução determinístico que nos permite repetir tudo de forma completa.

Registo de mensagens e consistência

- Quando é que devemos registar mensagens (log messages)?
  - Evitar processos órfãos:
    - Processo Q acaba de receber m_1 e m_2
    - Assumindo que a m_2 não foi registada.
    - Após a entrega de m_1 e m_2, Q envia mensagem m_3 ao processo R
    - Processo R recebe e subsequentemente entrega m_3: é um órfão.



Esquemas de Registo de Mensagens

- Notações
  - **DEP(m)**: processos aos quais foram entregues m. Se a mensagem m* é causada pela entrega de m, e m* foi entregue a Q, então $Q \in DEP(m)$.
  - **COPY(m)**: processos que contêm uma cópia de m, mas que não armazenaram (ainda) de forma fiável.
  - **FAIL**: conjunto de processos que crasharam
- Caracterização
  - Q é órfão $\Leftrightarrow Q \ni DEP(m)\ e\ COPY(m) \subseteq FAIL$

Esquemas de Registo de Mensagens

- Protocolo pessimista
  - Para cada mensagem instável m, existe no máximo um processo dependente de m, tal. Que $|DEP(m)| \leq 1$.
  - Consequência
    - Uma mensagem instável num protocolo pessimista tem que estabilizar antes de ser enviada a próxima mensagem.
- Protocolo optimista
  - Para cada mensagem instável m, garante-se que se $COPY(m) \subseteq FAIL$, então eventualmente também $DEP(m) \subseteq FAIL$
  - Consequencia
    - Para garantir que $DEP(m) \subseteq FAIL$, geralmente é feita a reversão de cada processo órfão Q até que $Q \notin DEP(m)$