

Tolerância a Falhas

THE LAST THANKSGIVING



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'. *Frontend depende do Backend ...*

Requisito	Descrição
Disponibilidade	Prontidão para ser utilizado <i>→ continuamente em funcionamento</i>
Fiabilidade	Disponibilização continua de um serviço
Segurança	Baixa probabilidade de <u>catástrofes</u> <i>⇒ bugs/pérdida de dados...</i>
<u>Manutenibilidade</u>	<u>Quão fácil é reparar o sistema</u> <i>Maintenability ... (montar e software) (reboot e dé fix...)</i>

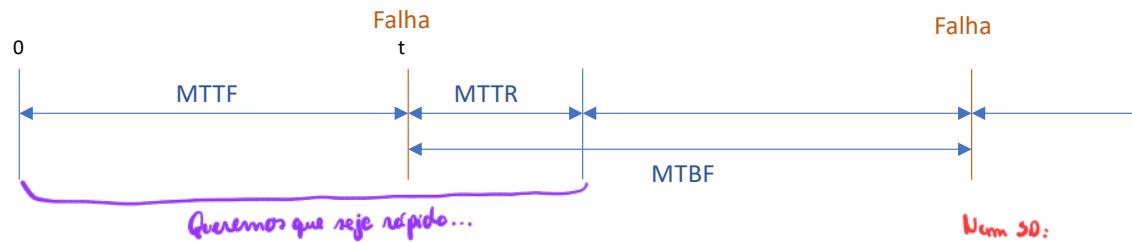
Definem
a confiança
no sistema ...

→ Corres que baseiam-se em software têm bugs
< fiabilidade

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 $t = 0$
→ Continua de funcionamento!
- 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

usados em SSDs
ele vai falhar... mas quanto tempo leva para corrigir...
→ Programa ④ por um maior nível de fiabilidade!



Nova ③:
→ BD corrompeu
→ reiniciar o serviço (MTTR)
→ Medimos os logs por exemplo...

Fiabilidade vs Disponibilidade

- Disponibilidade $D(t)$ de um componente C
 - Fracção média de tempo em que C esteve a funcionar no intervalo $[0, t[$
 - Disponibilidade a long prazo D: $D(\infty)$
 - Nota: $D = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF+MTTR}$
Disponibilidade ...
- 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 <u>não está a cumprir com as suas especificações</u> <i>causa do Failure! „</i>	Programa Crasha
Erro <i>no teste user!</i>	<u>Parte do programa que pode causar uma falha</u>	Bug
Falha/Culpa (Fault)	<u>Causa do erro</u> <i>Por de troç do erro...</i>	<u>Programador</u> Hardware, Catastrofe, variação de energia

Terminologia

Como prevenir

Termo	Descrição	Exemplo
Prevenção de Falhas ④ Servidores (não ser errado)	Prevenir a ocorrência de uma falha	Não contratar programadores desleixados (Metodologias, garantir que o código está bem feito...), teste exhaustivo
Tolerância a Falhas	Construir um componente tal que possa mascarar a ocorrência de falhas	Cada componente é desenvolvido independentemente por dois programadores → Códigos corretos de zero !!!
Remoção de Falhas	Reducir a presença, número, ou seriedade das falhas	Despedir maus programadores ④ Testes... Tenho servidores diferentes (quanto mais cedo eu... para o resto...) NÃO QUERO QUE SE ESPALHE!
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 Recursos Humanos Fazer versão Beta... → Quando está a chover vamos ter ④ falhos...

(Falhas = Fault)
culpa associada!

Modelos de Fracasso (Failure)

erro

Problemas em produção!

Tipos de fracassos	Descrição do comportamento do servidor
Crash fracassado	Pára, mas funciona correctamente até parar
Omissões <ul style="list-style-type: none">- Omissão na recepção- Omissão no envio	Não responde a mensagens Não recebe novas mensagens Não envia mensagens
Falha temporal	Responde fora dos limites temporais especificados.
Falha na resposta <ul style="list-style-type: none">- Falha no valor- Falha transição de estado	Resposta é incorrecta O valor da resposta está errado Desvio do controlo de fluxo correcto
<u>Falha Arbitrária</u>	Pode produzir respostas arbitrárias em tempos arbitrários

→ valores arbitrários que não deviam ter acontecido!

Confiança vs Segurança

Falha por omissão: Não fez!
Falha por comissão: Fez mal!

- Omissão vs Comissão

- Uma falha arbitrária pode ser classificada como maliciosa. → Eventuais danos no Sistema...
- **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
 - ↳ Tem de ter requisito...
 - ↳ Do que em sistema chegam lá...

Difícil distinguir

- **ATENÇÃO**: uma falha deliberada é tipicamente um problema de segurança. Distinguir entre umas e outras é no entanto geralmente impossível.

Falhas terminais (Halting failures)

Falha terminal!

- 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 processo ou tempos de entrega de mensagens → **não se pode detectar falhas por crash de forma fiável.**
*Ele está a falhar, mas será que ele morreu?
→ vai depender...*
 - **Sistema síncrono:** **tempos de execução e entrega de mensagens estão delimitados** → **podemos detectar de forma fiável falhas por omissão 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**

No projeto ~

↳ 10 em 10 segundos tento de receber um HELLO!

Falhas terminais

• Consistência dos dados

Tipo de paragem	Descrição
Fail-stop	Falha por crash, mas detectada fiavelmente
Fail-noisy	Falha por crash, <u>eventualmente detectável</u> (<i>streaming com pixels desfocados</i>)
Fail-silent	Falhas por omissão ou crash: cliente não distingue o que aconteceu
Fail-safe	Arbitrária, mas <u>benigna</u> (i.e. não fazem estrago)
Fail-arbitrary	Arbitrária, com <u>falhas maliciosas</u> <i>(uma exceção do código... é só o bom que podemos testar-las)</i>

try
catch →

Redundância para mascarar falhas

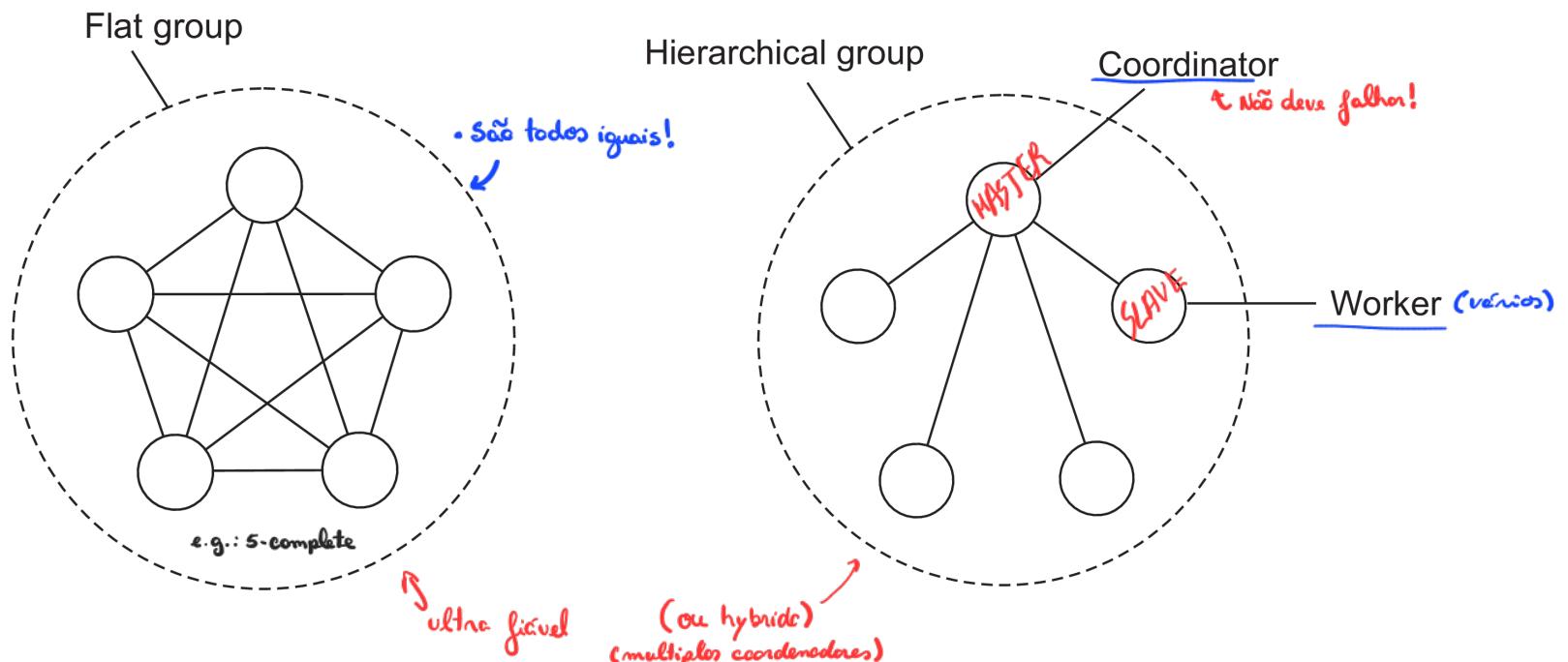
- Tipos de redundância

- Informação: Acrescentar bits extra às unidades de dados para que os erros possam ser recuperados quando os bits são danificados \Rightarrow código corretor de erro ... { um ficheiro para corrigir os erros...}
- 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. \leftarrow TCP , tentamos de novo & funciona
- Física: Acrescentar equipamentos ou processos por forma a permitir que um ou mais componentes possam falhar. Este tipo é usado extensivamente em sistemas distribuídos. \oplus Nodes... (redundância física)

\rightarrow informação : \oplus bits
 \rightarrow temporal : \oplus tentativas
 \rightarrow física : \oplus nós

Resiliência Processo

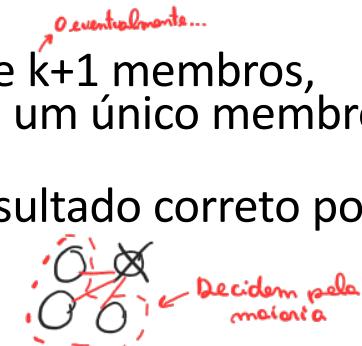
- Proteger contra o mau funcionamento de um processo através da replicação de processos, organizando múltiplos processos em grupos de processos. Distinguir entre grupos planos e grupos hierárquicos



Grupos e máscaras de falha

- **K-fault tolerant group**

- Quando um grupo consegue mascarar quaisquer falhas de membros concorrentialmente (k é chamado de grau de tolerância a falhas)
- O quanto 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 **arbitrarias**: necessário $2k+1$ membros para que o resultado correto possa ser obtido por maioria de votos.
 - Democracia ... tem de ser impar > 3*
 - consenso entre os membros ...*
- 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.*With K-fault tolerance group
they can still execute commands in the same order ...*
- Reformulação
 - Membros do grupo que não falham precisam de chegar a um consenso sobre qual comando executar de seguida

Consenso por flooding

Bom para o Projeto Final! //

- Modelo de Sistema

- Considere um grupo de processos $P = \{P_1, \dots, P_n\}$
- Semântica de falha Fail-stop, isto é com detecção fiável de falhas.
todos detectam a falha
- 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 Cmd_i^r para todos os outros.
- No final de r , cada P_i funde todos os comandos recebidos num novo Cmd_i^{r+1} .
- Próximo comando Cmd_i é selecionado por uma função determinística ?

④ Pequeno para maior
④ Simples para mais...

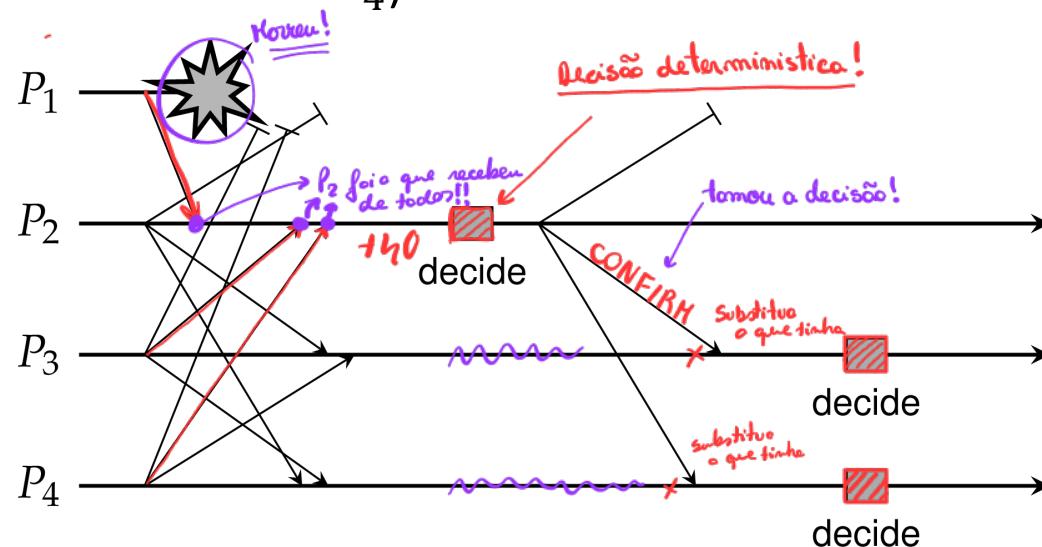
OBRIGATÓRIO
é escolhido um comando
apenas no final de cada ronda!

1 processo não pode dar essa ordem...

Consenso por flooding: Exemplo

Todos têm de conhecer todos! // condição necessária!

- P_2 recebe todos comandos proposto por todos outros processos \rightarrow 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 $P_2 \rightarrow$ não pode tomar decisão (mesmo acontece com P_4)



• P_2 recebeu de todos!
tipicamente num flat-group
Desvantagens:
→ Todos têm de saber tudo!
→ Temos de periodicamente enviar comandos vazios...
→ Muitos pacotes

ZooKeeper é um sistema distribuído que garante que todos os processos se conhecem!

Consenso Realista: Paxos

Provado!

• Uma versão realista é o RAFT!
Consenso realista

→ Consenso com base em pre requisitos

Implementação é difícil

- Assunções (fracas, e realistas)

- Sistema é **parcialmente síncrono** (na realidade até pode ser assíncrono)
- **Comunicação** entre processos pode ser **instável**: mensagens podem ser perdidas, duplicadas ou reordenadas ↪ *Não podem ser maliciosos!*
- **Mensagens corrompidas podem ser detectadas** (e subsequentemente ignoradas)
Se tivermos outro algoritmo para tal!
- Todas **operações são determinísticas**: uma vez iniciada uma execução, é sabido o que irá acontecer.
- Processos podem exibir **falhas por crash**, mas **não falhas arbitrárias.** ↪ *maliciosos!*
- **Processos não podem conspirar.** ↪ *Valores fora do protocolo!,,*

→ Paper de 1998 de Leslie Lamport

↓
Só pode
mover!,,

X

Problema dos impérios bizantinos: dependem de mercenários e para executar um ataque é preciso agir todos os mercenários. Todos têm de concordar, e tem de ser apartir de mensagens... (perdemos um mensageiro...)
Eles não podem conspirar! //

NÃO é o PAXOS

Apenas uma dedução!!

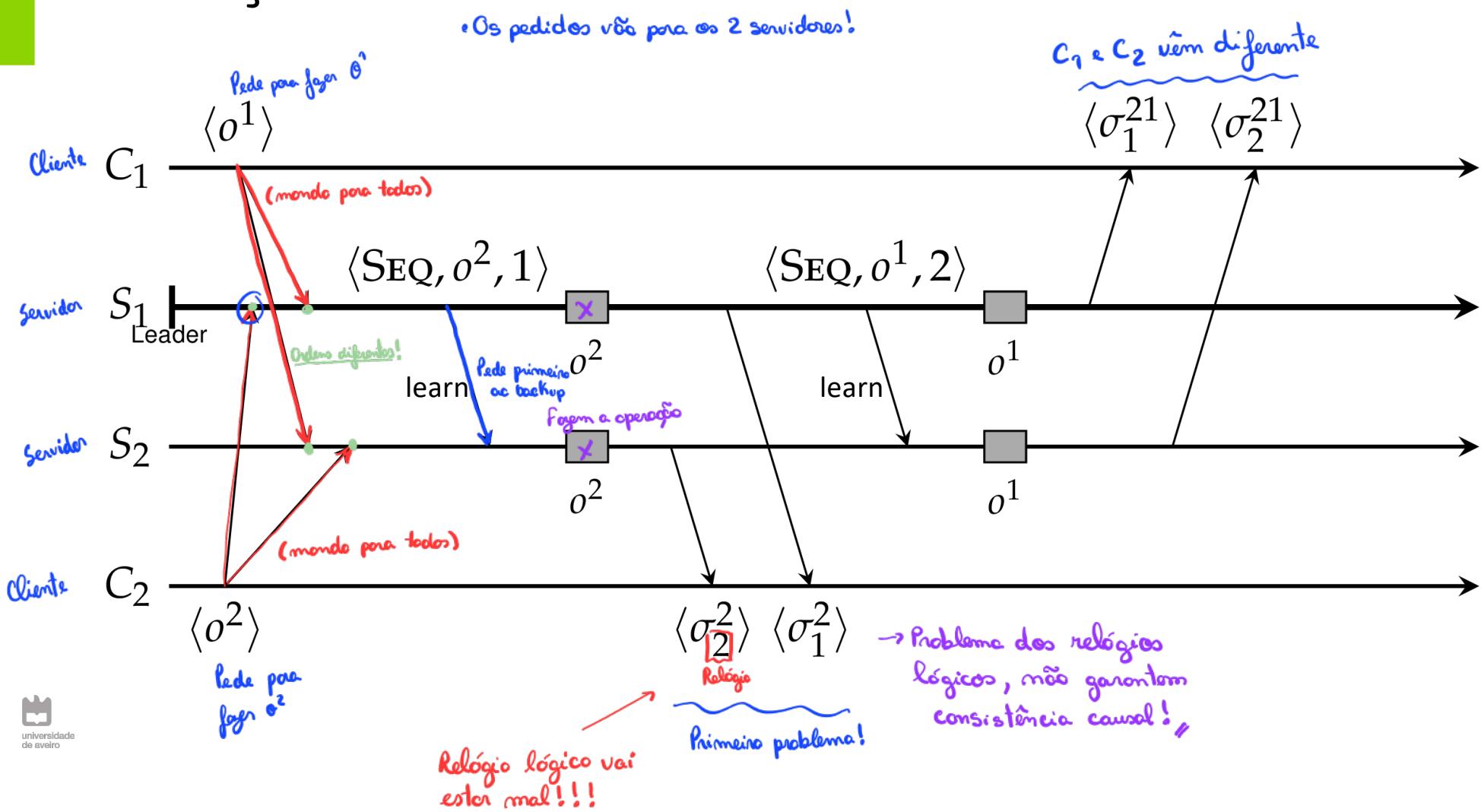
Dedução do algoritmo Paxos

- É assumida uma configuração cliente-servidor, inicialmente com um servidor primário ← Líder!
- Para tornar o servidor mais robusto, começamos por adicionar um servidor de backup. ← Servidor de backup! } Mas precisa de estar sincronizado com o Líder Queremos Fault-Tolerance!
- Para garantir que todos os comandos são executados pela mesma ordem em ambos os servidores, o primário atribui números de sequência únicos a todos os comandos. No Paxos o primário é chamado de líder.
Relógio Lógico
Todos os operações estão ordenadas! → Operações ordenadas (relógio lógico)
Secundário segue o primário!!
- Assumir que os comandos actuais podem sempre ser restaurados (tanto através dos clientes ou dos servidores) → apenas consideradas mensagens de controlo.

Primário faz 100x e manda para o Backup...

X → Vai haver uma sequência de slides com erros propositados!

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.

Pedido

Pedido e confirmação

Aprendi no tempo t ,
e Aprendi

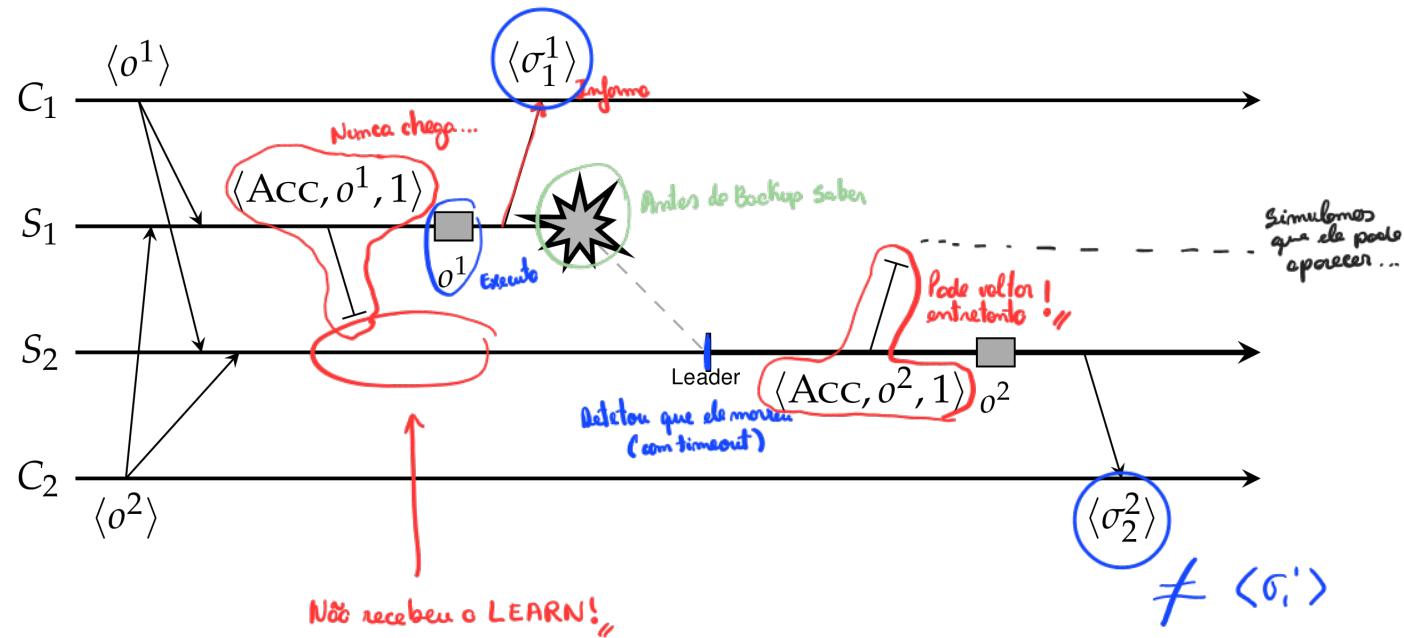
Se o primário
cracha ...





Dois servidores e um crash: problema

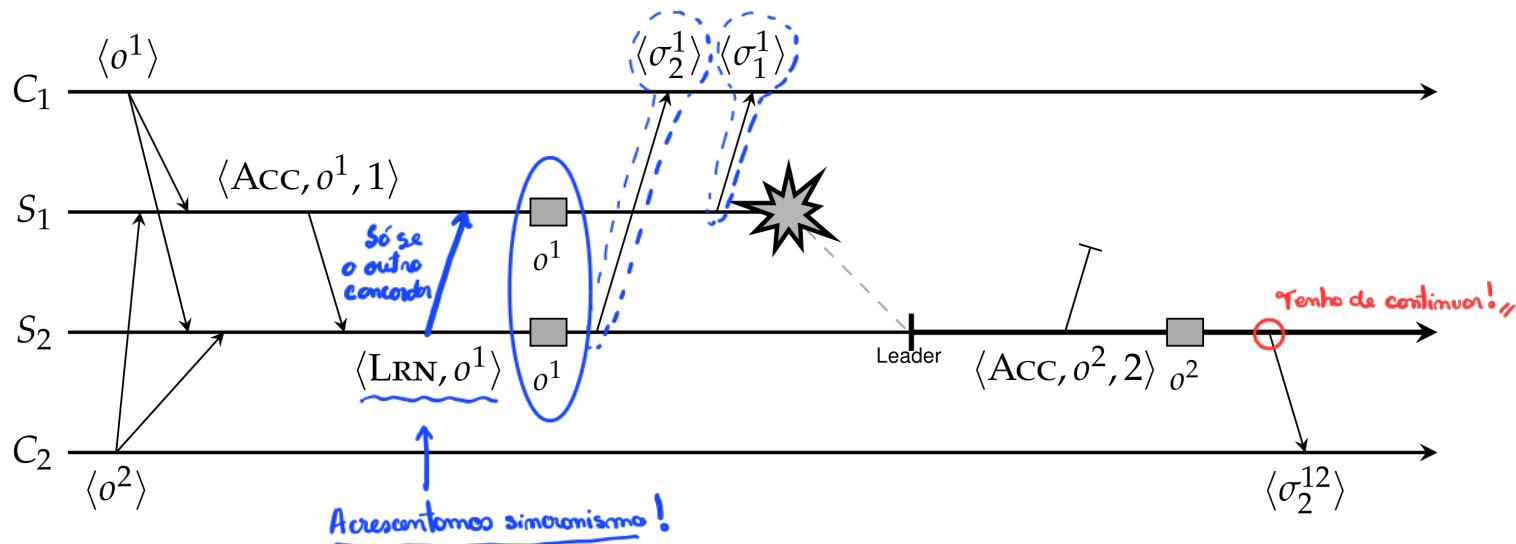
- Primário crasha após executar uma operação, mas o backup nunca recebeu a mensagem de aceitação.





Dois servidores e um crash: solução

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



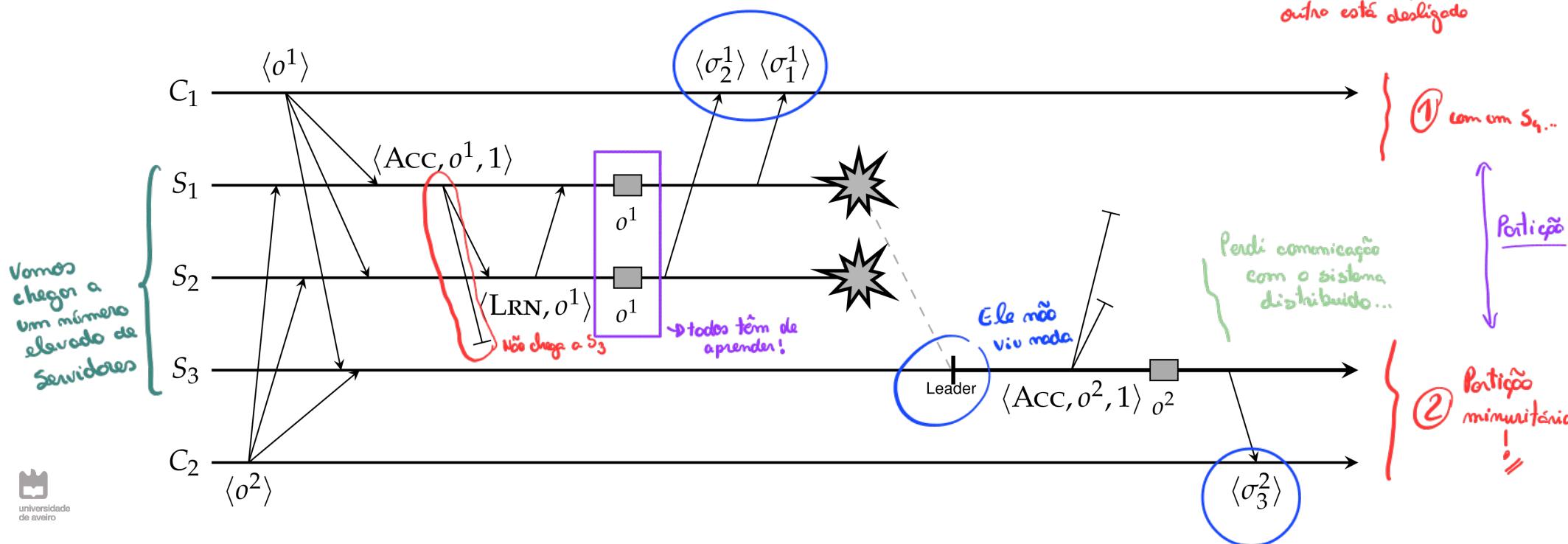
X

Três servidores e dois crashes: problema?

Que acontece quando $\text{LEARN}(o^1)$ é enviado de S_2 para S_1 e é perdido?

S_2 necessita esperar que S_3 tenha aprendido o^1

Entrou em partição
cada lado pensa que o outro está desligado



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

↑ Parte síncrona!



X

Detecção de falhas

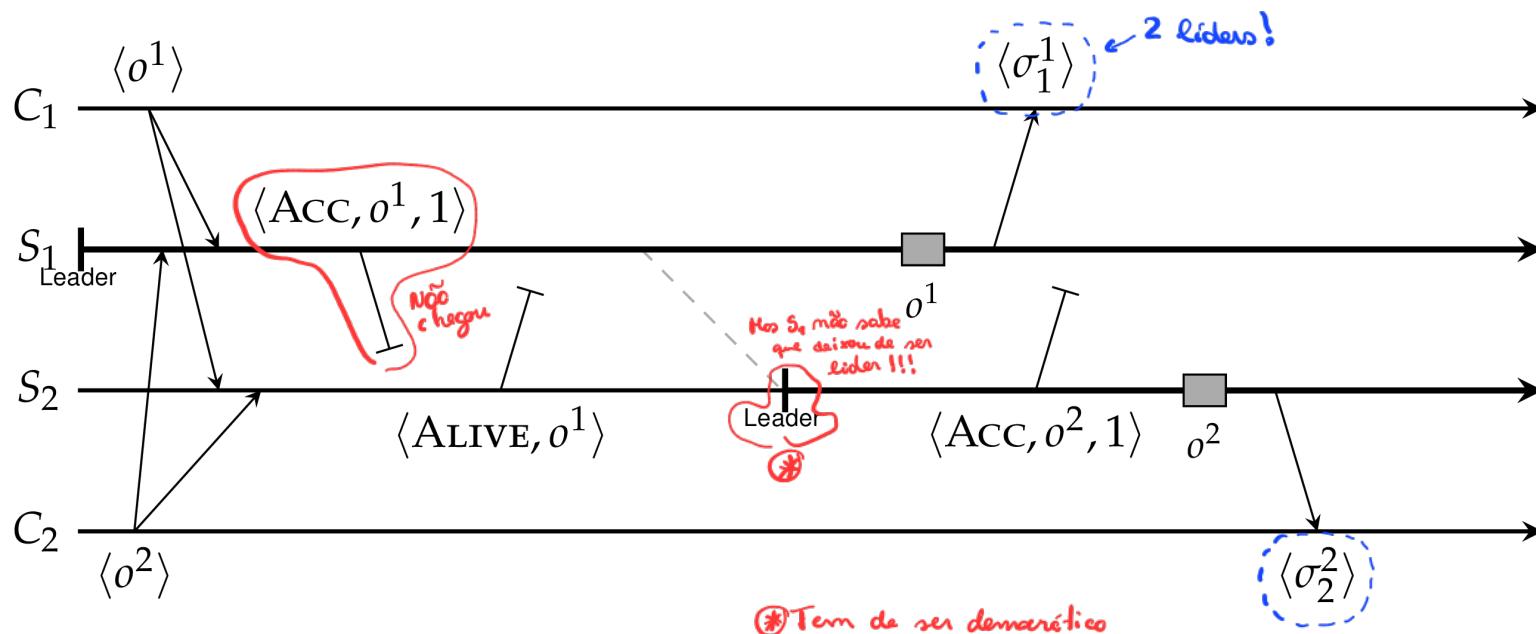
• Quais são os não operacionais?

chave...
8 //

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

→ Ou estão atrasados...

Veio atrasado mas está tudo OK! //



PAXOS

Número de Servidores necessários

Paxos

- 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.

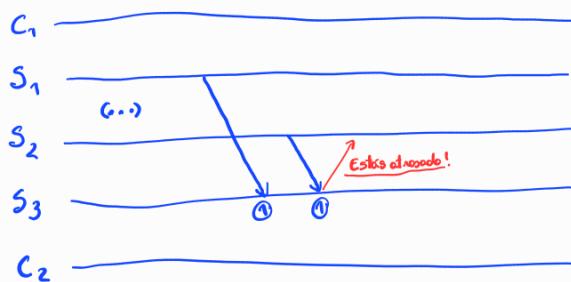
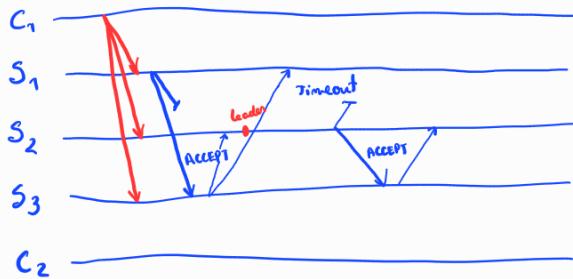
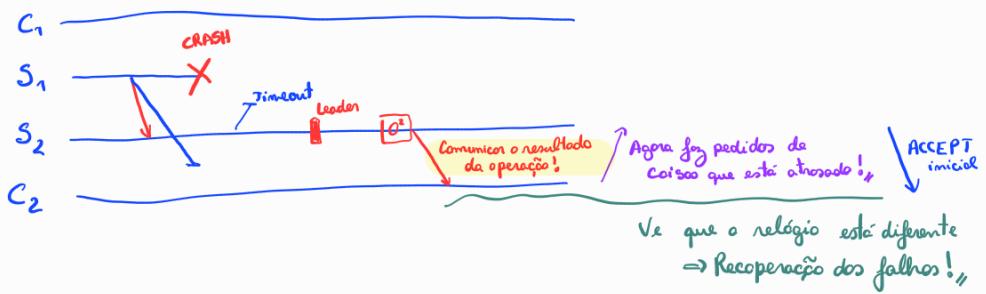
Para apenas 1 crash! $2K+1$



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($o^2, 2$) $\rightarrow S_3$ vê um timestamp inesperado e informa S_2 que perdeu o^1
 - S_2 retransmite ACCEPT($o^1, 1$)
 - *Manda uma confirmação!* $\rightarrow S_3$ recuperar
- S_2 não recebe ACCEPT($o^1, 1$)
 - S_2 detectou o crash e torna-se no leader
 - S_2 envia ACCEPT($o^1, 1$) $\rightarrow S_3$ retransmite LEARN(o^1)
 - S_2 envia ACCEPT($o^2, 2$) $\rightarrow S_3$ informa S_2 que aparentemente perdeu ACCEPT($o^1, 1$) de S_1 pelo que S_2 pode recuperar

X



↓ Próx. slides



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.

Apenas $\gamma_{//}$



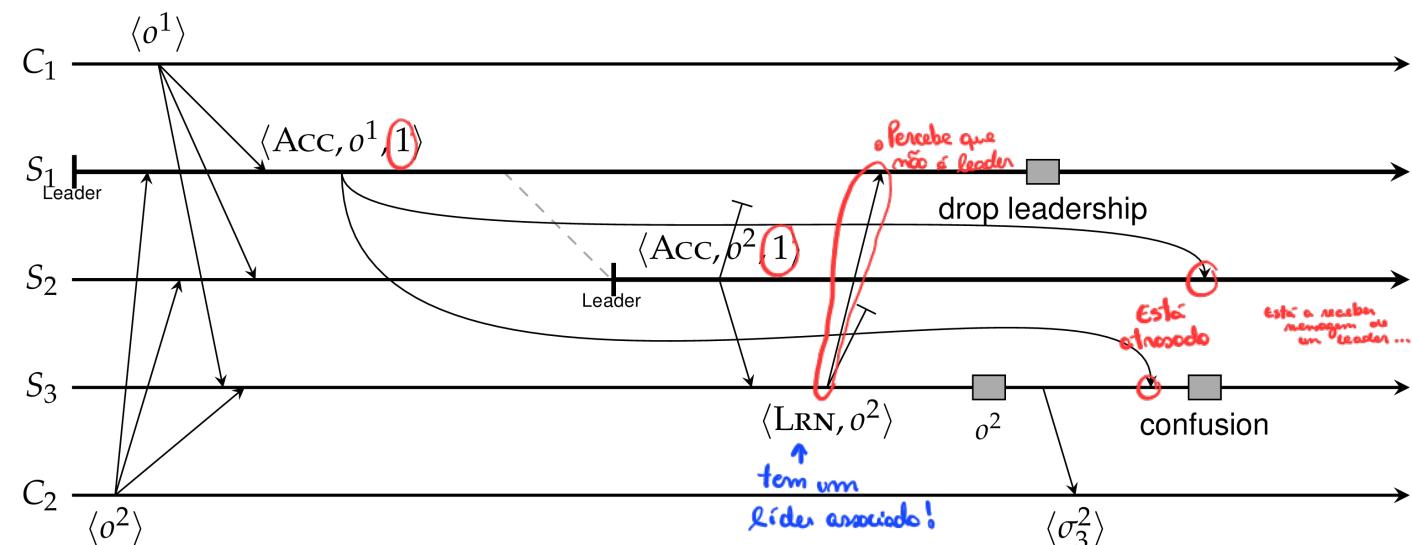


Detecção de falsos crashes

- S_3 recebe um $\text{ACCEPT}(o^1, 1)$, mas muito depois de $\text{ACCEPT}(o^2, 1)$. Se soubesse quem é o leader actual, poderia rejeitar a mensagem atrasada.

- Leaders devem incluir o seu ID nas mensagens

Relógio lógico deve ter desempate!
(relógio deve ser incrementado pelo líder!)



Paxos Operação Normal

Paxos define 3 papéis: **proposers**, **acceptors** e **learners**

P1a – Prepare \Rightarrow todos usam o mesmo relógio
(definido por quem propõem a operação)
Ancorar timestamp

P1b – Promise
“Fecha” o timestamp
Promoço que vamos cumprir

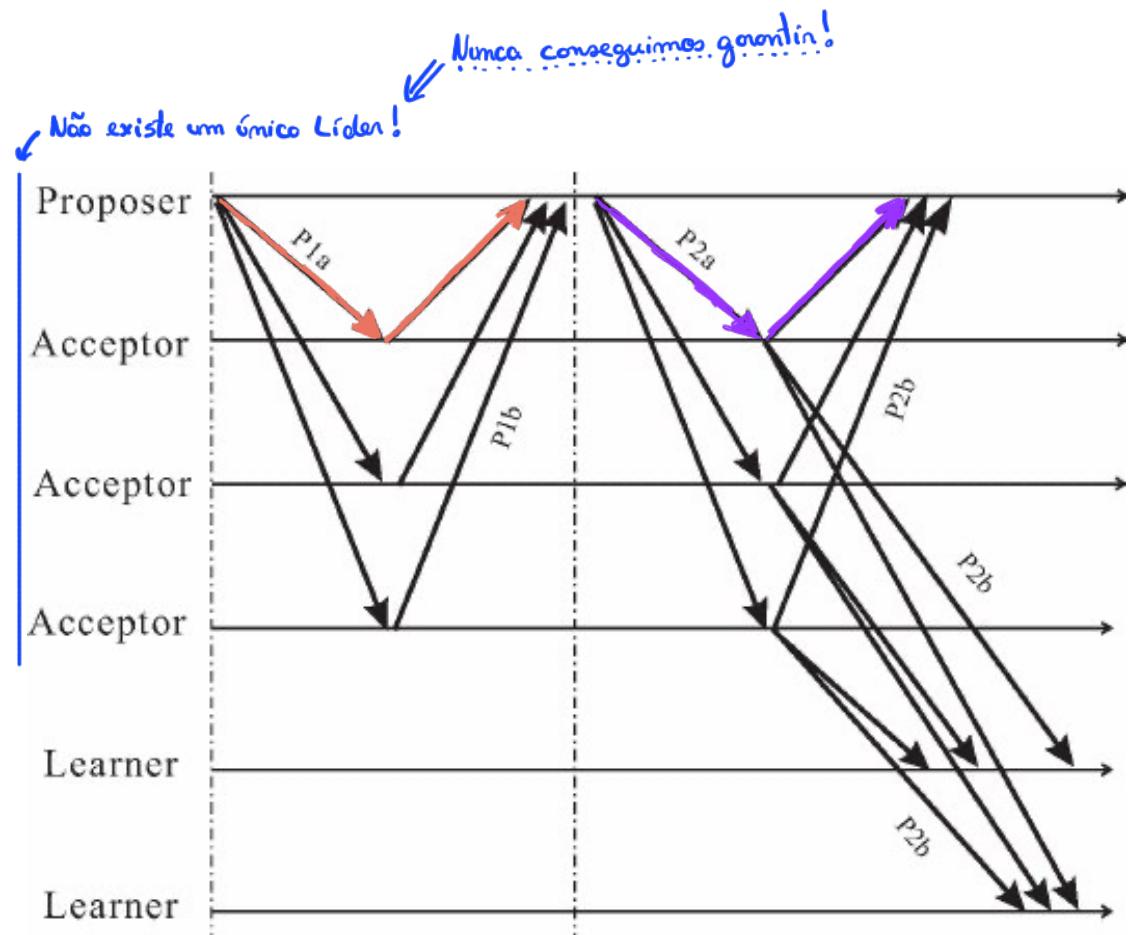
P2a – Accept \Rightarrow OK envia!
“envia o seu valor”

P2b – Learn
“executa operação”

O mesmo servidor normalmente tem os 3 estados

“The paxos Algorithm” – está no elearning.

MUITO BOM!

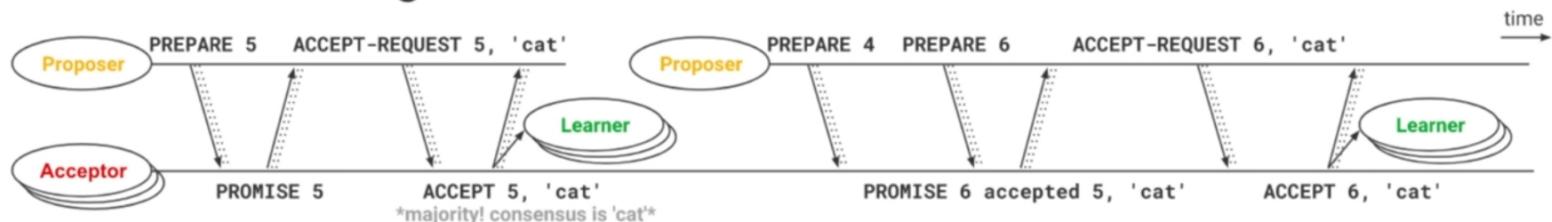


Propose \rightarrow : toda a gente vai usar o mesmo relógio, se definido por quem propõem a operação, ele dig “Sincronizem os relógios que vamos trabalhar nest”

Resposta: Promise

Accept \rightarrow Resposta: Learn
(para todos)

The Paxos Algorithm



- ⇒ **Proposer** wants to propose a certain value:
It sends **PREPARE ID_p** to a majority (or all) of **Acceptors**.
ID_p must be unique, e.g. slotted timestamp in nanoseconds.
e.g. **Proposer 1** chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) ID_p.
- ⇒ **Acceptor** receives a **PREPARE** message for ID_p:
Did it promise to ignore requests with this ID_p?
Yes -> then ignore
No -> Will promise to ignore any request lower than ID_p.
Has it ever accepted anything? (assume accepted ID=ID_a)
Yes -> Reply with **PROMISE ID_p accepted ID_a, value**.
No -> Reply with **PROMISE ID_p**.
- 💡 If a majority of acceptors promise, no ID<ID_p can make it through.**

- ⇒ **Proposer** gets majority of **PROMISE** messages for a specific ID_p:
It sends **ACCEPT-REQUEST ID_p, VALUE** to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes -> It picks the value with the highest ID_a that it got.
No -> It picks any value it wants.
- ⇒ **Acceptor** receives an **ACCEPT** message for ID_p, value:
Did it promise to ignore requests with this ID_p?
Yes -> then ignore **LEARN**
No -> Reply with **ACCEPT ID_p, value**. Also send it to all **Learners**.
- 💡 If a majority of acceptors accept ID_p, value, consensus is reached.**
💡 Consensus is and will always be on value (not necessarily ID_p)
- ⇒ **Proposer** or **Learner** get **ACCEPT** messages for ID_p, value:
- 💡 If a proposer/learner gets majority of accept for a specific ID_p, they know that consensus has been reached on value (not ID_p)**

Google



Consenso em falhas arbitrárias

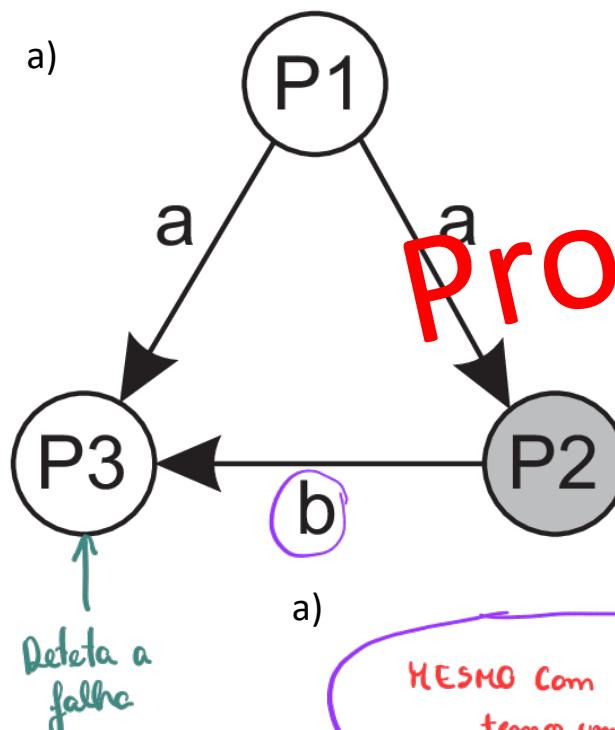
Corrompidos, ou errados!

Paxos não assegura isto!

• Os primários não mantêm!

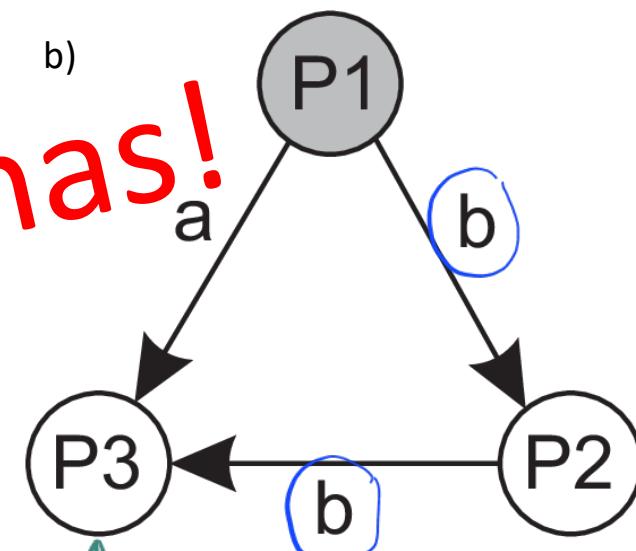
- Considere grupos de processos em que a comunicação entre processos é inconsistente: (a) mau encaminhamento de mensagens ou (b) comunicar coisas diferentes a diferentes processos.

Mensagem corrompida
Hackers!



Problemas!

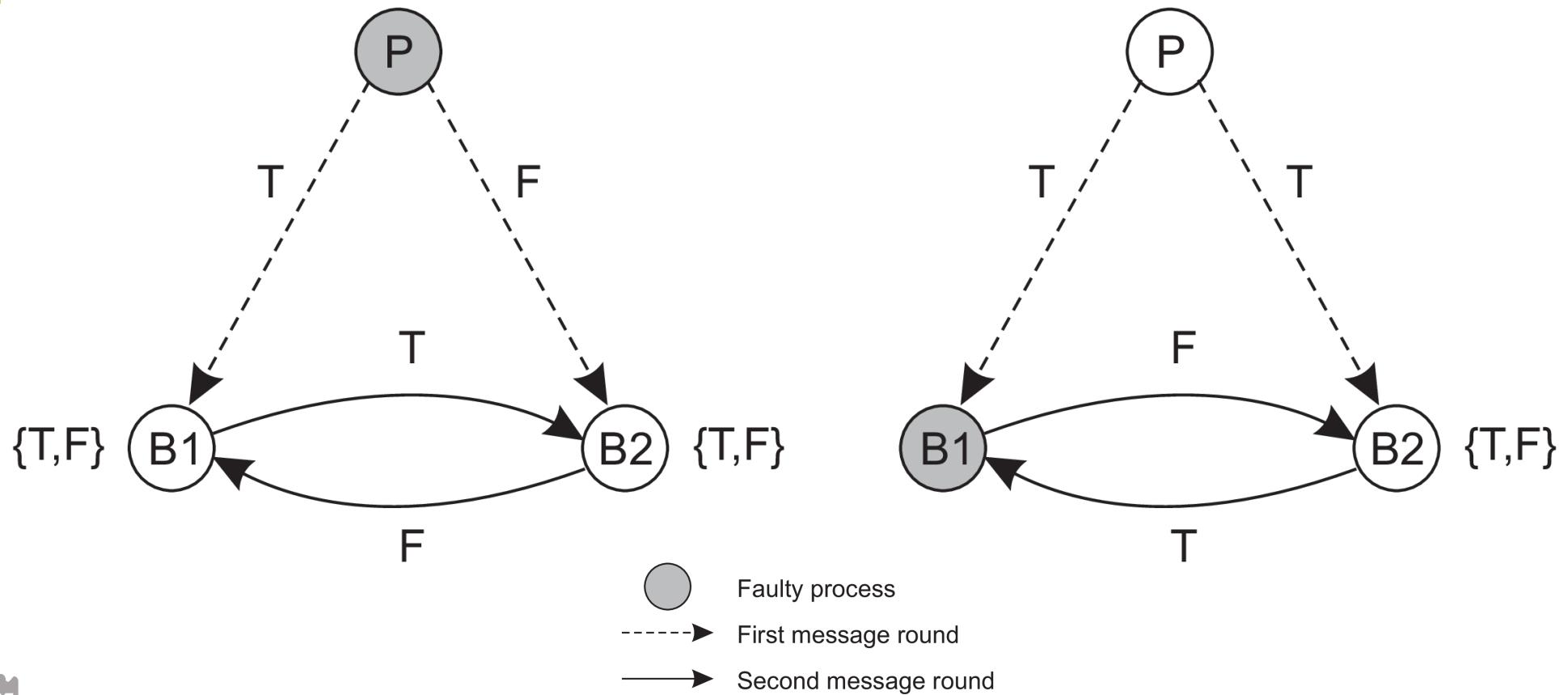
MESMO com números ímpares
temos um problema
!!!



Consenso em falhas arbitrárias

- Modelo Sistema
 - Consideramos um primário P e $n-1$ backups B_1, \dots, B_{n-1}
 - Um cliente envia $v \in \{T, F\}$ a P
→ Por um id nos mensagens!
 - Mensagens podem se perder, mas tais são detectadas
 - Mensagens não podem ser corrompidas sem haver detecção
→ Assinatura criptográfica!
 - O receptor de uma mensagem detecta fielmente que foi o emissor
→ Com a assinatura!
- 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.
O primário é de confiança

Porque ter $3k$ processos não é suficiente



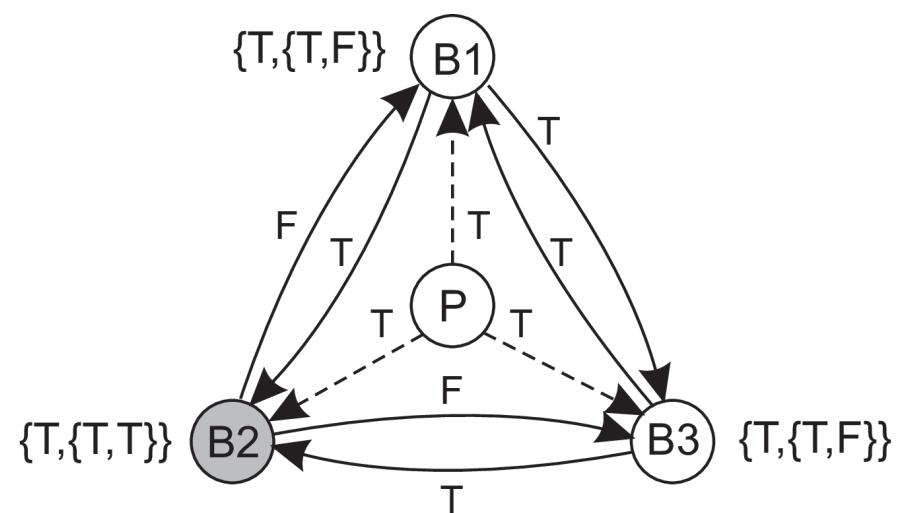
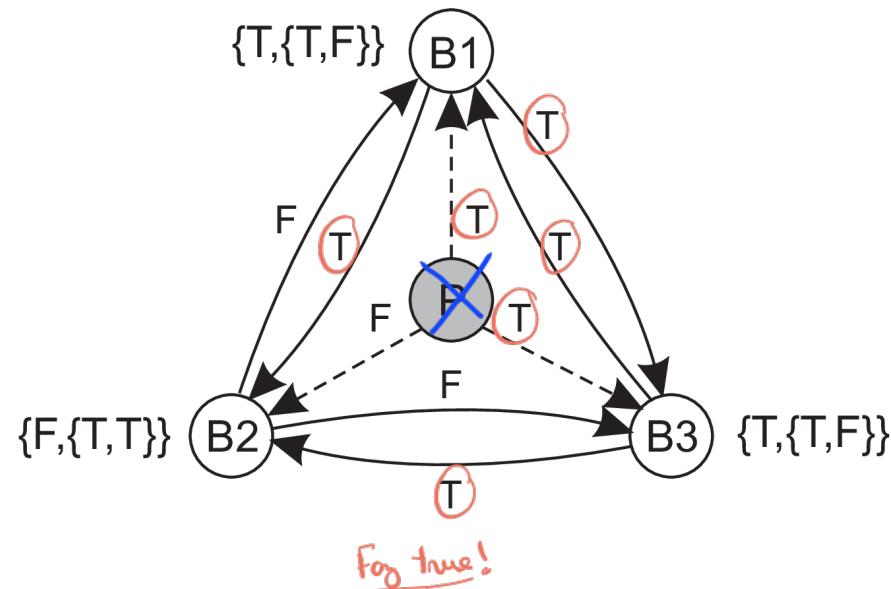
- Assumimos que o hacker não entra em 2 servidores ao mesmo tempo



Com uma única falha!

Porque ter $3k+1$ processos é suficiente

Para manter uma maioria!



- Faulty process
- > First message round
- > Second message round

Concretizar tolerância a falhas

Se um Datacenter arder não pode queimar todos
↓ Availability Zones !

Na cloud ele em princípio está distribuído por Datacenters !



- Considerando que os membros de um grupo tolerante a falhas encontram-se fortemente acoplados, podemos-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 ?
 - E o que acontece quando um grupo se encontra particionado ?

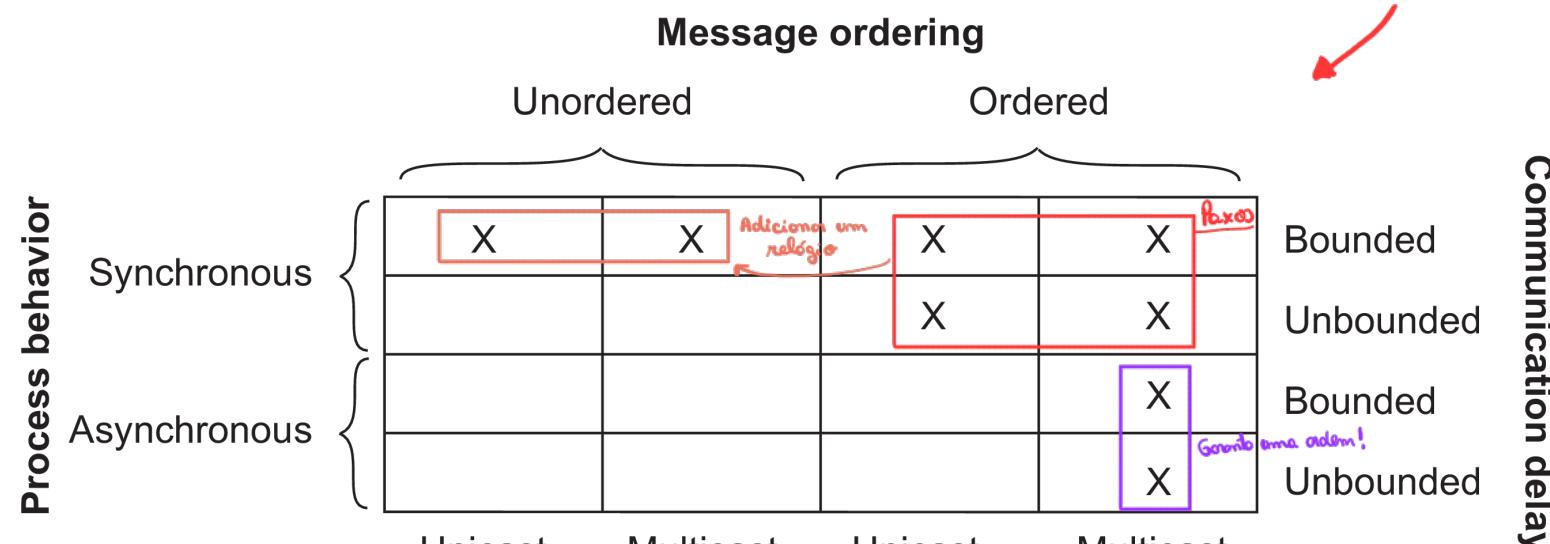
→ Protocolo
→ Não é eficiente
→ $3, 5, 7, \dots, 2K+1, K > 0$
→ Com falhos arbitrários $3K+1$!
Só para sistemas de riscos !

É difícil que os sistemas
sejam tolerante a falha ...

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

Pergunta de Teste !



	Unsorted	Sorted	
Sync	X X	X X	Bounded
		X X	Unbounded
Async			Bounded
			Unbounded
	Unicast	Multicast	
	Unicast	Multicast	

Consistência, disponibilidade e particionamento

*Numa rede
as 3*

MongoDB (não tem consistência)



• Teorema CAP

essencial a base de dados!

- Qualquer sistema distribuído que partilhe informação só pode ter duas das seguintes três propriedades:
 - Consistência, qualquer informação partilhada e replicada aparece com um único valor actualizado
 - Availability (Disponibilidade), que garante que uma actualização será sempre eventualmente executada
 - Partição 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.

Availability

"temos de abdicar de uma!"

• Por baixo de uma BD temos um sistema distribuído
→ construído com 2 das propriedades CAP:
→ Consistency
→ Availability
→ Partition tolerant

Partition tolerant

Consistency

Projeto Final:

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 intervalo t : P suspeita de Q
deteção de timeout
- Se Q mais tarde enviar uma mensagem (que é recebida por P):
 - P deixa de suspeitar de Q
 - P aumenta o valor do intervalo t
- Atenção: se Q tiver crashado, P irá continuar a suspeitar de Q .

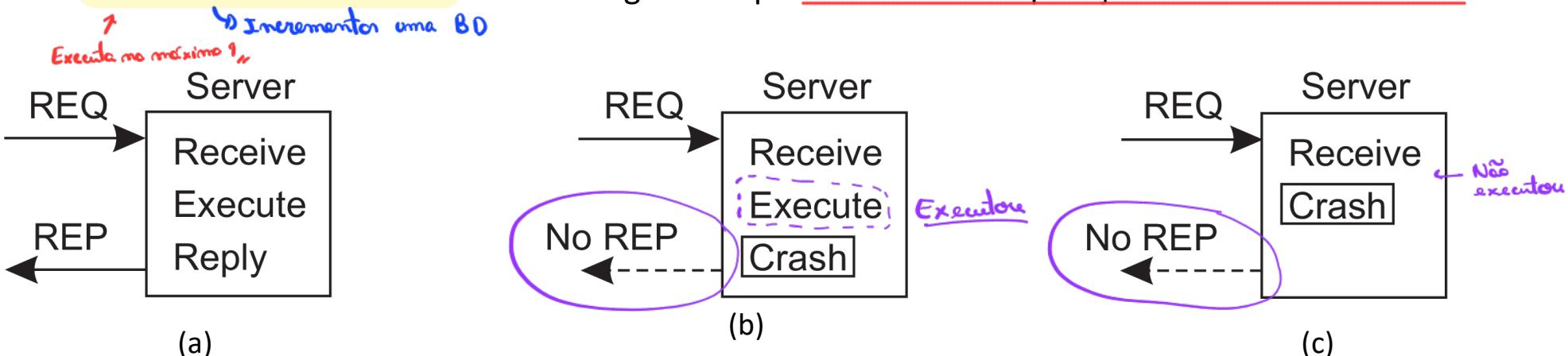
RPC's fiáveis

- 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 crasha 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
 - Enquanto (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.



Porque é impossível recuperar de uma falha no servidor transparentemente ?

- Três eventos distintos no servidor
 - M: envio de uma mensagem completa
 - P: completa o processamento de um documento
 - C: crash
- Seis ordens diferentes
 - $M \rightarrow P \rightarrow C$: Crash depois de reportar término
 - $M \rightarrow C \rightarrow P$: Crash depois de reportar o término, mas antes de actualizar
 - $P \rightarrow M \rightarrow C$: Crash depois de reportar o término, e depois de actualizar
 - $P \rightarrow C (\rightarrow M)$: Actualização teve lugar, e depois crashou
 - $\boxed{C}(\rightarrow P \rightarrow M)$: Crash antes de qualquer acção
 - $\boxed{C}(\rightarrow M \rightarrow P)$: Crash antes de qualquer acção

Nunca recebo
nem processso

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 ^{duplicação}	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

• Não existe solução perfeita! „

RPC fiável: perda de mensagens

- O que o cliente repará, 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)
Reformular requisições
 - 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 operações são idempotentes por natureza, tais como as transacções bancárias.

→ Podem ser aplicados várias vezes

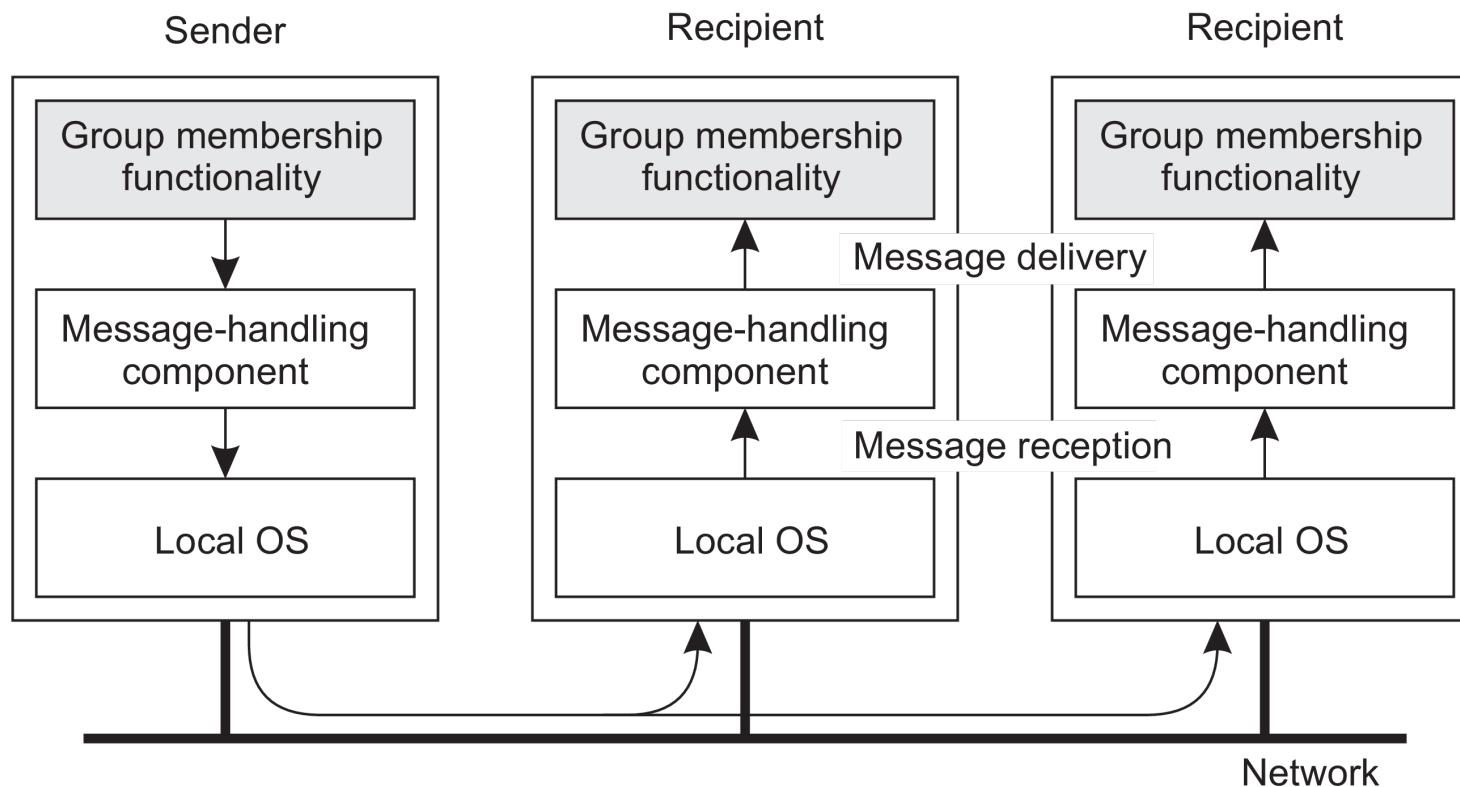
RPC fiável: cliente crasha

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

RPC para além de 1:1 (cliente - Servidor)

Comunicação em grupo (multicast) fiável e simples

mas sempre é possível...

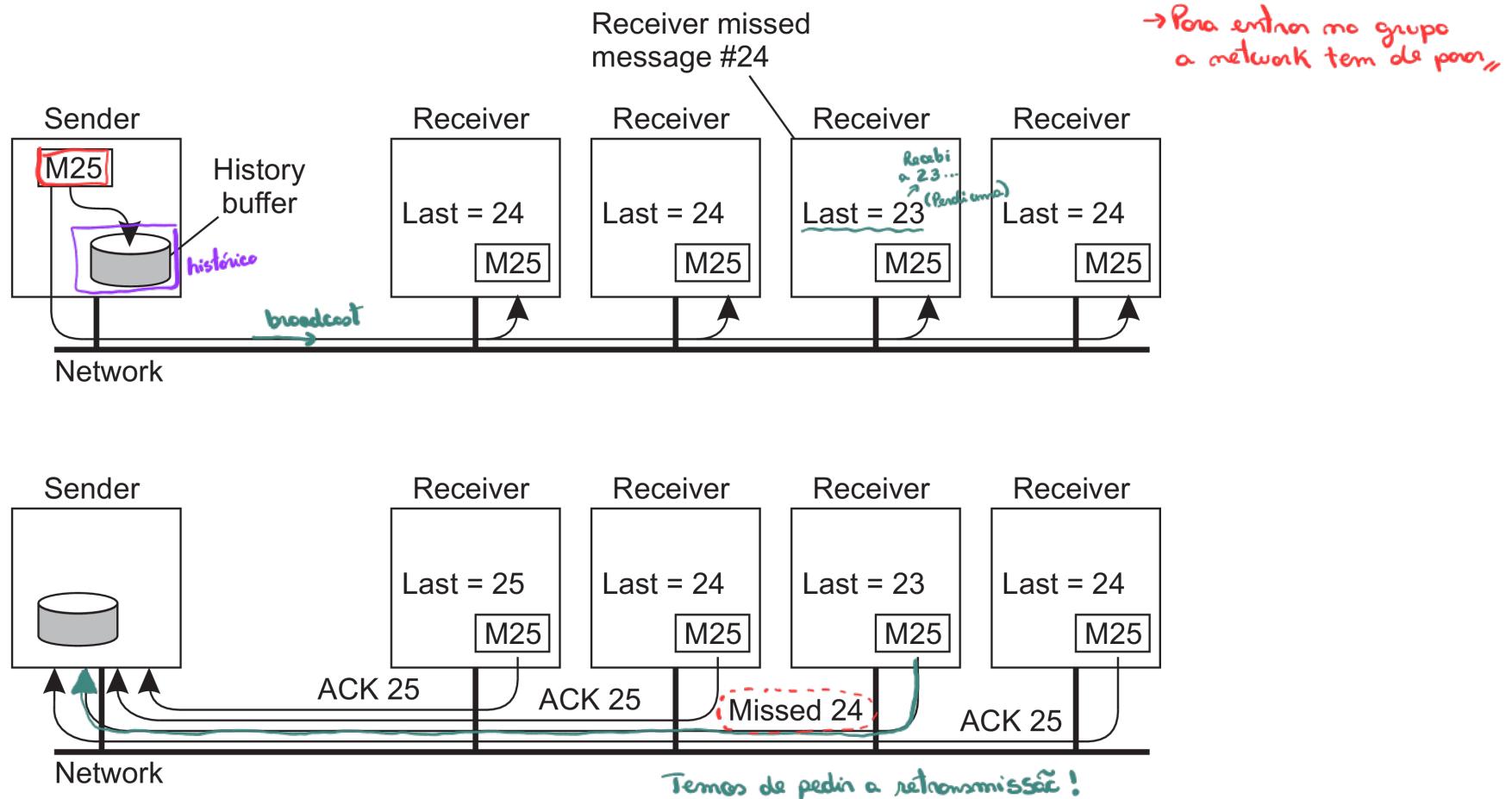


Comunicação em grupo fiável (não simples)

• Como sei toda a gente?

- 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 membros 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

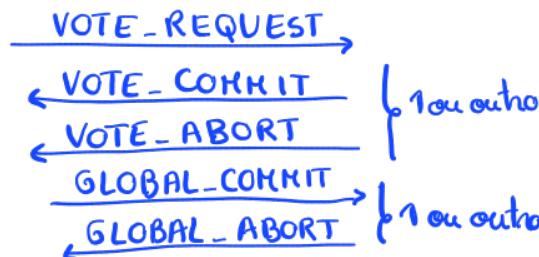
- Problema

- Garantir que uma operação é executada por todos membros de um grupo, ou que é executada por nenhum (tudo ou nada) //

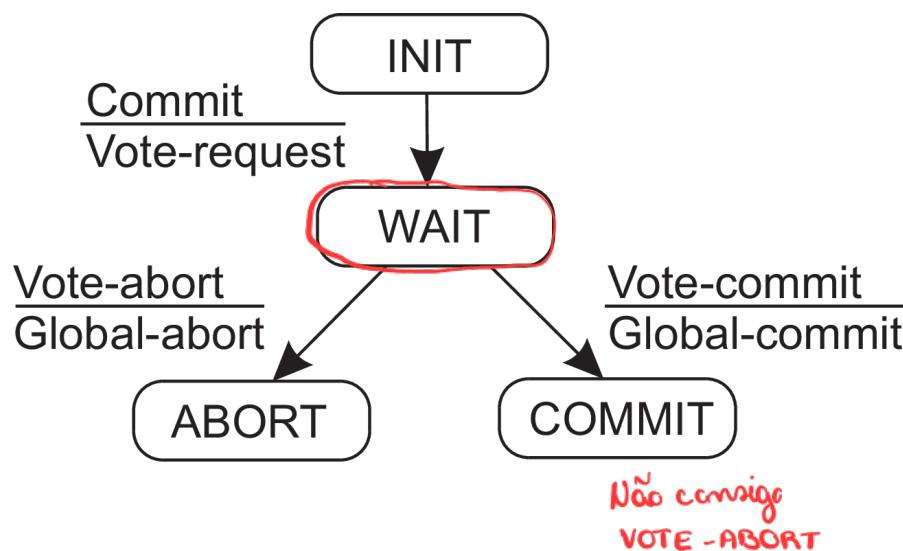
- Multicast fiável: uma mensagem é entregue a todos os receptores
 - Transacção distribuída: cada transacção local tem que ter sucesso

Protocolo de commit em 2-passos (2PC) ← Projeto Final!

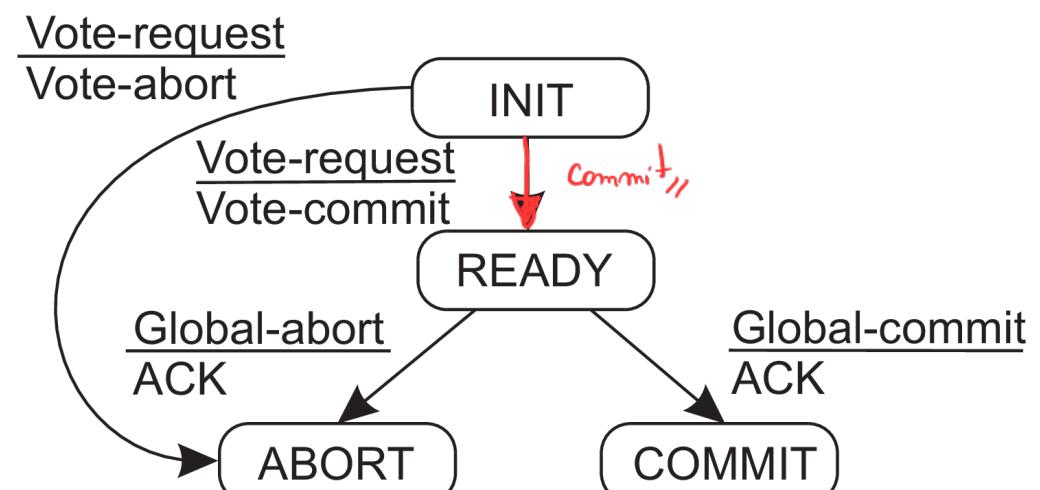
- O cliente que inicia a computação age como coordenador; processos que necessitam tomar parte do commit são participantes.
 - Fase 1a: Coordenador envia VOTE-REQUEST aos participantes (também chamado de **pre-write**)
Vote request
 - Fase 1b: Quando participante recebe VOTE-REQUEST responde com VOTE-COMMIT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



2PC – Máquinas de estados finitos



Coordenador



Participante

2PC – Participante falha

- Participante crasha no estado S , e recupera para S
 - INIT: Não há problema, participante não conhecia o protocolo
 - 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 pode tornar o estado de commit idempotente, isto é, copiar o workspace para storage Já recebeu commit final...
- 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 poder 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. *Detetar falhas do coordenador!*
- Um participante não pode decidir localmente, depende sempre dos outros processos (eventualmente em falha)

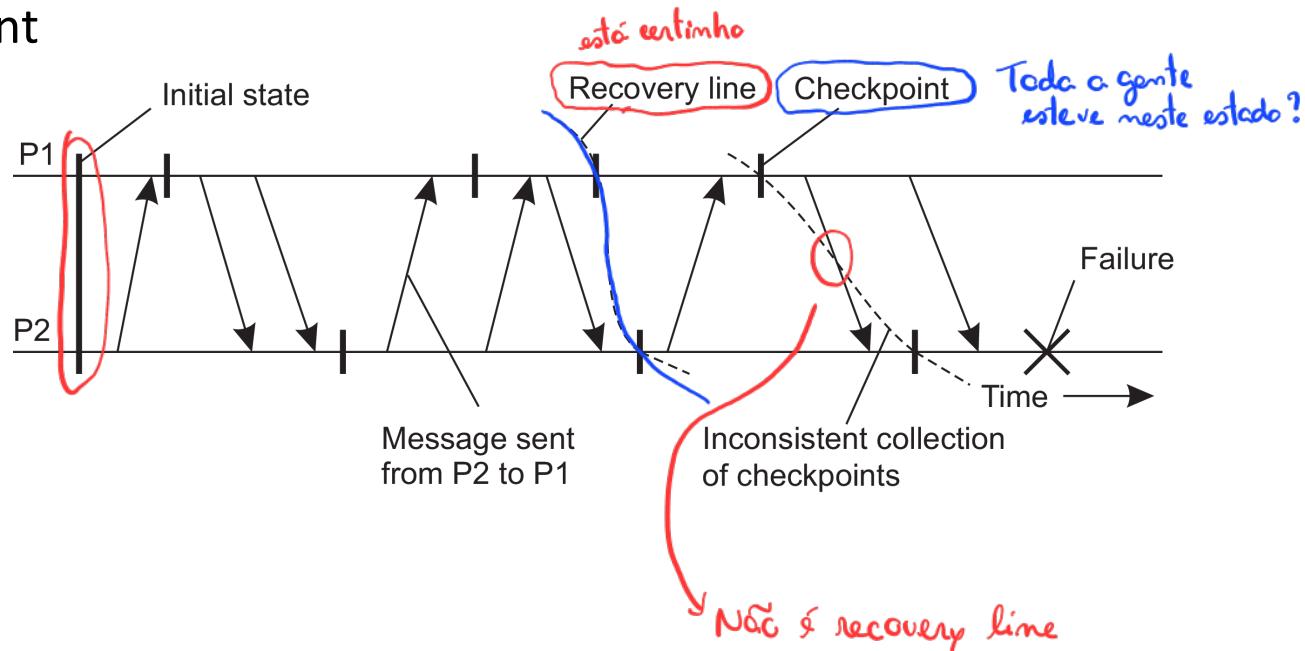
*Abortar commit
eleger candidatos*

Recuperação de Falhas

- Quando uma falha acontece, é necessário trazer o processo de volta a um estado sem falhas:
 - Avançar no erro (Forward Error Recovery): Encontrar um novo estado em que o Sistema possa **continuar** a operar
 - *Fuga para a frente*
 - "Perdi? OK, segue..."* (transmissão de vídeo)
 - Estado novo consistente no tempo*
 - Retroceder no erro (Backward Error Recovery): Voltar **atrás** no Sistema a um estado livre de erros
 - *Fuga para trás*
 - "Perdi? Voltar para algo que estava bem"*
- **Backward Error Recovery** é o mais usual, mas necessita do estabelecimento de pontos de recuperação (**recovery points**) ☺
- A recuperação em sistemas distribuídos é complicada pelo facto dos processos terem necessidade de cooperar na identificação de um **estado consistente**, a partir do qual possam recuperar

Recuperação de estado consistente

- Requisito
 - Para cada mensagem que tenha sido recebida é demonstrável 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) → Bloqueia a aplicac... (Não podemos abusar...)
 - É possível nos cingirmos apenas aos processos que dependem da recuperação do coordenador e ignorar os demais. Os de leitura não precisam...

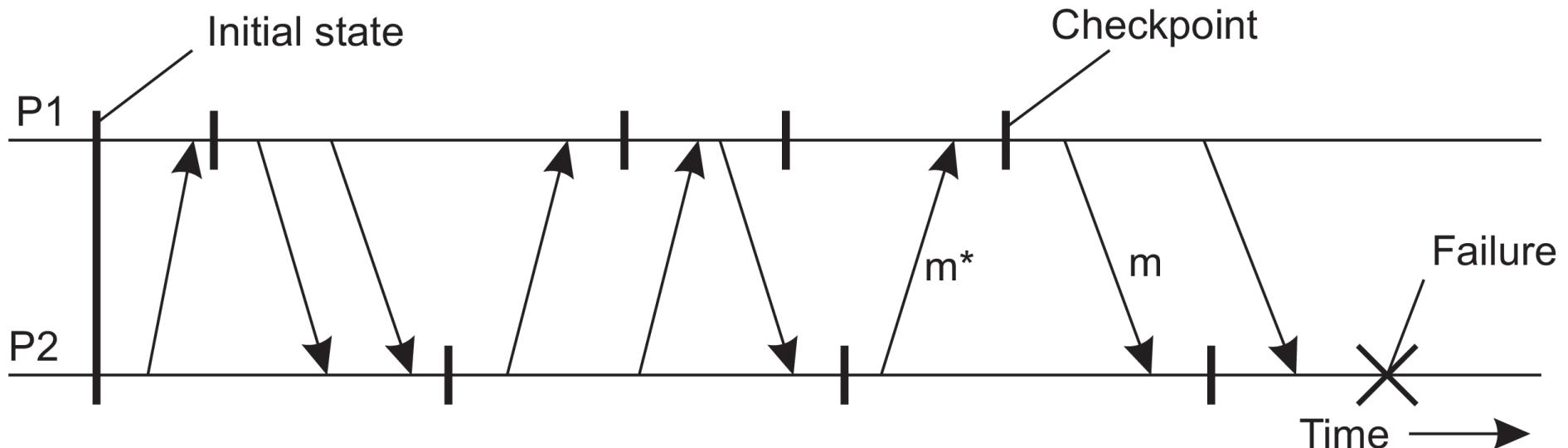
Reversão em Cascata

O pior caso é começar do zero!



- 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”

Vamos ver se os checkpoints estão consistentes



Checkpoints independentes

checkpoints independentes

- Cada processo cria checkpoints de forma independente, com o risco de uma reversão em cascata.
 - Seja $CP_i(m)$ o m^{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 $CP_j(n - 1)$
 - Quando forem múltiplos são verificados

Sincronização
os checkpoints

Alternativa aos checkpoints !

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.

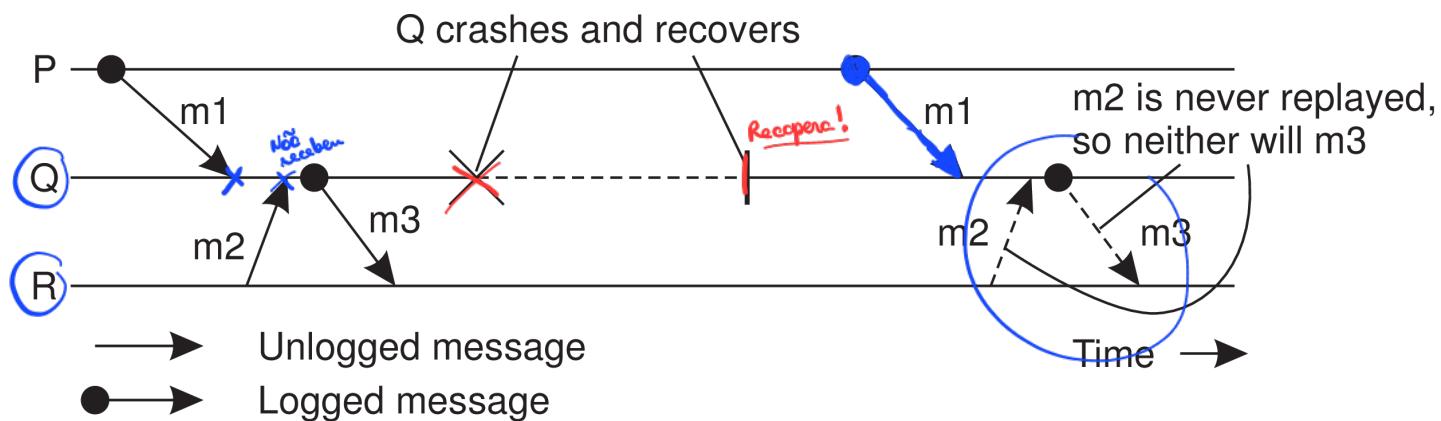
• Commits muito pequenos
com snapshots ...

Registo de mensagens e consistência

- Quando é que devemos registar mensagens (log messages)?

- Casos de 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 $\Leftrightarrow Q \not\in DEP(m) \text{ e } \underline{COPY(m) \subseteq FAIL}$

Esquemas de Registo de Mensagens

- Protocolo pessimista
 - Para cada mensagem instável m , existe no máximo um processo dependente de m , tal. Que $\underline{|DEP(m)|} \leq 1$.
 - 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 $DEP(m) \subseteq FAIL$
 - Consequência
 - Para garantir que $DEP(m) \subseteq FAIL$, geralmente é feita a reversão de cada processo órfão Q até que $Q \notin DEP(m)$