

Tolerância a Faltas

Modelo de Interação: sistemas síncronos vs. assíncronos

- Sistema **síncrono** é aquele em que são garantidos os seguintes limites:
 - Cada mensagem enviada chega ao destino dentro de um tempo limite conhecido
 - O tempo para executar cada passo de um processo está entre limites mínimo e máximo conhecidos
 - A taxa com que cada relógio local se desvia do tempo absoluto tem um limite conhecido
- Caso algum destes limites não seja conhecido, o sistema é **assíncrono**

Sistemas síncronos vs. assíncronos: implicações para os sistemas tolerantes a falhas

- Um nó N1 deixou de responder a outro nó N2
- N2 é capaz de detetar (com certeza) que N1 falhou?

Modelo de Faltas

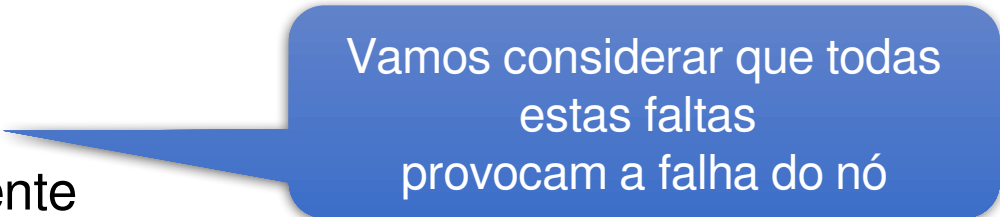
- No modelo é necessário identificar quais as faltas *expectáveis*
- Em seguida, decidir:
 - Quais as faltas que vão ser recuperadas
 - Quais as faltas que **não** vão ser toleradas
- A *taxa de cobertura* é a relação entre as faltas que têm possibilidade de ser recuperadas e o conjunto de faltas previsíveis
- As faltas que originam erros sem possibilidade de tratamento dão origem a *catástrofes*

Modelo de Faltas num Sistema Distribuído

- Num sistema distribuído o modelo de faltas é muito mais complexo que num sistema centralizado.

Várias componentes do sistema podem falhar:

- Faltas na comunicação
- Faltas nos nós
 - Processadores/Sistema
 - Processos servidores ou clientes
 - Meios de Armazenamento Persistente



Vamos considerar que todas estas faltas provocam a falha do nó

Tipos de Faltas

- Faltas silenciosas
 - Quando componente pára e não responde a nenhum estímulo externo:
 - Falta pode ser detetável (*fail-stop*) ou não detetável (*crash*)
- Faltas arbitrárias (bizantinas):
 - Quando qualquer comportamento do componente é possível:
 - É o pior caso possível
 - Útil para tolerar erros de *software* ou ataques

Faltas silenciosas dos processos

- Quando um processo em falha deixa de responder a estímulos do exterior

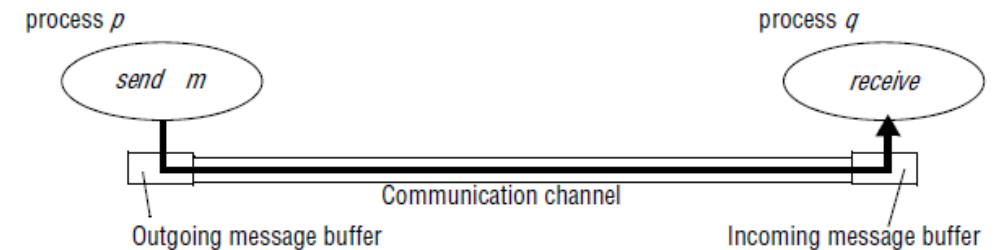


Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

If you'd like to know more, you can search online later for this error: HAL_INITIALIZATION_FAILED

Faltas silenciosas do canal

- Quando o canal não entrega uma mensagem
- Podemos distinguir entre:
 - *Send-omission*
 - Mensagem perdeu-se entre o processo emissor e o *buffer* de saída para a rede
 - *Channel-omission*
 - Mensagem perdeu-se no caminho entre ambos os *buffers*
 - *Receive-omission*
 - Mensagem chegou ao *buffer* de entrada do recetor mas perdeu-se depois



Faltas arbitrárias (bizantinas)

- De um processo
 - Processo responde incorretamente a estímulos
 - Processo responde mesmo quando não há estímulos
 - Processo não responde a estímulos
- Do canal de comunicação
 - Mensagem chega com conteúdo corrompido
 - Entrega mensagem inexistente ou em duplicado
 - Não entrega mensagem
- Faltas arbitrárias do canal de comunicação são raras pois os protocolos sabem detetá-las
 - Como? O que fazem quando as detetam?

Por omissão, assume-se falta silenciosa

- Por simplificação, é muitas vezes assumido que a falta é silenciosa sem que haja real demonstração que é assim
- No tipo de faltas que é vulgar não considerar no subconjunto a recuperar temos:
 - Faltas densas
 - Faltas bizantinas
- Ao pressupor um modelo de faltas que não se verifique na realidade, um sistema desenhado para ser tolerante a faltas pode não cumprir a sua especificação

Faltas frequentemente não consideradas no modelo

Falta densa

- Acumulação de tantas faltas toleráveis que deixa de ser tolerável

Falta arbitrária (bizantina)

- Faltas que fogem ao padrão de comportamento especificado para a componente, por exemplo, um nó da rede que envia mensagens corretas a um interlocutor e erradas a outro

Deteção de faltas num sistema distribuído

Deteção num Sistema **síncrono**

Assume-se a existência de uma latência máxima entre nós da rede (bem conhecida) e um tempo máximo de processamento de cada mensagem

Deteção num Sistema **assíncrono**

Mais realista, e.g., durante uma **partição na rede**, ou ataque **Denial-of-Service**

Não é possível limitar a latência da rede ou o tempo de resposta do servidor

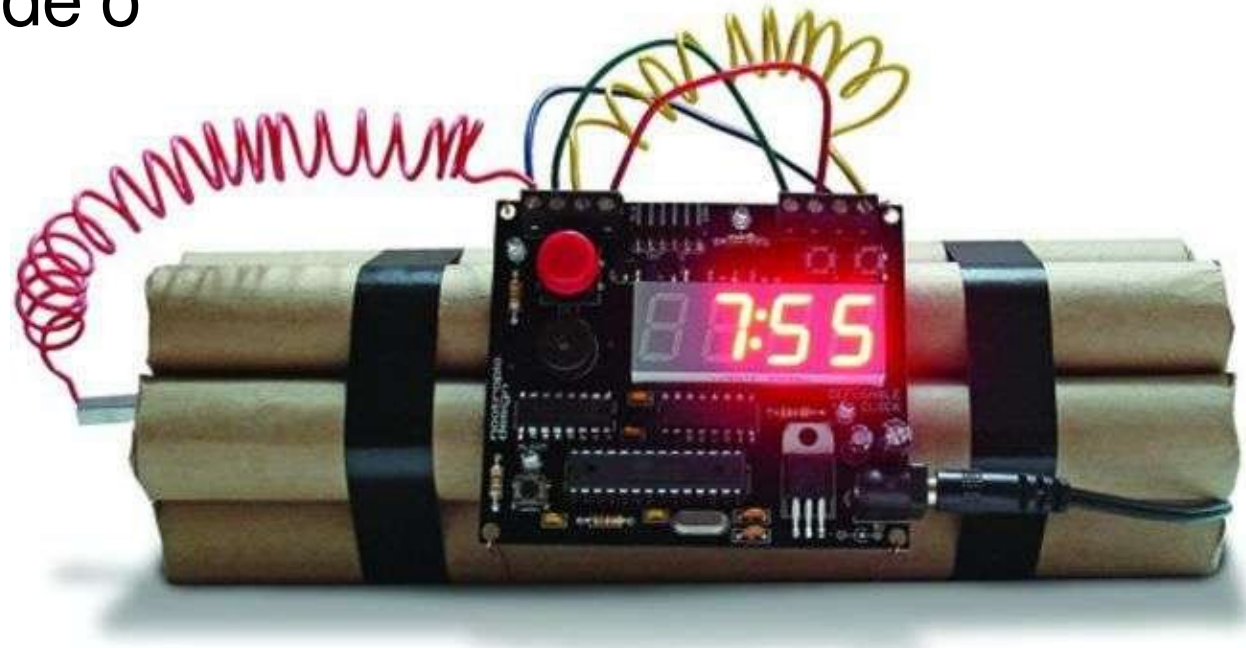
É impossível a deteção remota de falhas por paragem. Pode ser confundida com um aumento na latência

Como quantificar quão tolerante a faltas é um sistema?

Medidas usadas em tolerância a faltas

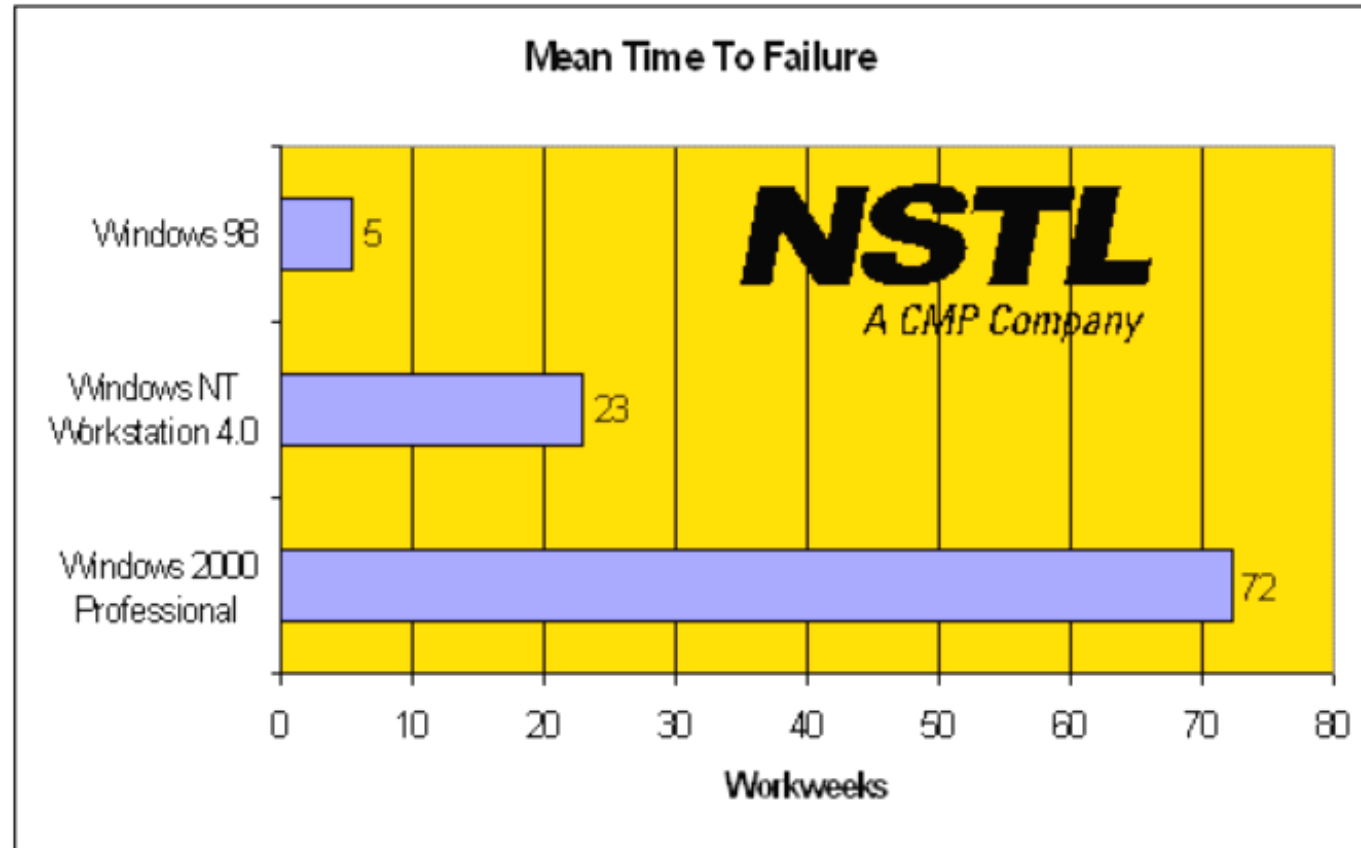
Métrica: fiabilidade (*reliability*)

- Mede o tempo médio desde o instante inicial até à falha

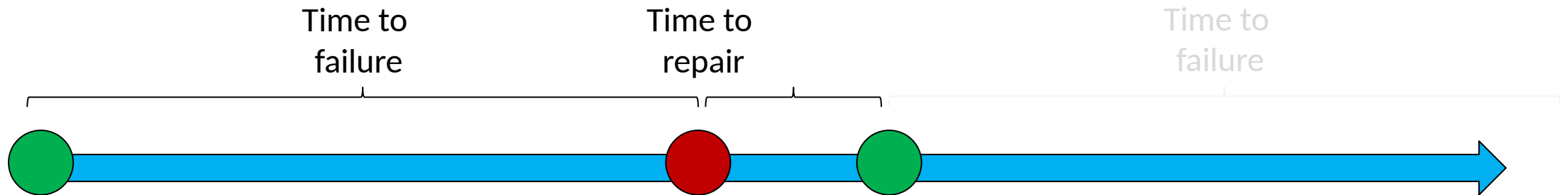


- Para sistemas não reparáveis, chamamos *Mean Time To Failure* (MTTF)

Exemplo MTTF para medir fiabilidade de sistema operativo



Métrica: fiabilidade em sistemas reparáveis



- Em sistemas **reparáveis**, fiabilidade é normalmente dada pelo *Mean Time Between Failures (MTBF)*
 - Tempo médio desde reinício correto até à próxima falha


Exemplo MTBF para medir fiabilidade de cartão de memória

amazon.co.uk
Computers & Accessories industrial sd card

Deliver to Portugal Shop by Department Your Amazon.co.uk Today's Deals Gift Cards & Top Up Hello, Sign in Your Account

Computers & Accessories Best Sellers Deals Laptops Desktops Printers Tablets Tablet Accessories Monitors

Back to search results for "industrial sd card"



Transcend TS128MSD100I Industrial SD CARD
by Transcend
★★★★★ 1 review from Amazon.com

Price: £19.44 & FREE UK Delivery on orders dispatched by Amazon over £20. [Delivery Details](#)

In stock.

Want it delivered by Thursday, 19 Apr.? Order within 5 hrs 46 mins and choose Expedited Delivery at checkout. [Details](#)

Dispatched from and sold by Amazon. Gift-wrap available.

Note: This item is eligible for click and collect. [Details](#)

3 new from £19.40 2 used from £11.88

- Product description: Transcend sd100i
- Width: 2.4 cm
- Width: 2.1 mm (0.0827)
- Height: 3.2 cm
- Capacity: 0.125 GB
- Type of Memory Card: secure digital (SD)
- Read speed: 27 MB/s

Mean time between failures: 1000000h

- Protection functions: over shock resistant
- Case Material/Body: ABS plastic, polycarbonate
- Memory voltage: 2.7/2.8, 3.3, 3.6 V

[Show Less](#)

Exemplo: cálculo do MTBF

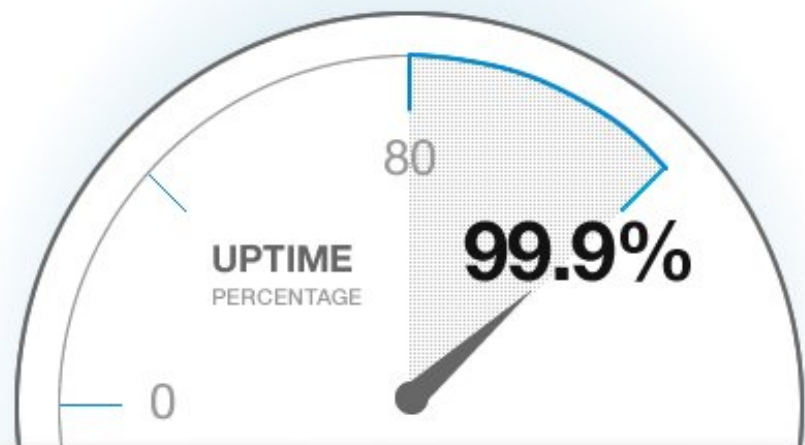
“Um serviço deveria operar corretamente durante 9 horas. Durante este período, ocorreram 4 falhas. O tempo de todas as falhas totalizou 1 hora.”

- Qual o valor de **MTBF** (tempo médio entre falhas)?

$$MTBF = (9-1)\text{horas} / 4 \text{ falhas} = 2 \text{ horas}$$

Métrica: disponibilidade (*availability*)

- Mede a relação entre o tempo em que um serviço é fornecido e o tempo decorrido



- **Disponibilidade** = $MTBF / (MTBF + MTTR)$
 - **MTTR** = Mean Time to Repair
 - **MTBF** = Mean Time Between Failures

Exemplo: cálculo do MTTR

*“Um serviço deveria operar corretamente durante 9 horas.
Durante este período, ocorreram 4 falhas.
O tempo de todas as falhas totalizou 1 hora.”*

- Qual o valor de **MTTR** (tempo médio para reparação)?

$$\begin{aligned} \text{MTTR} &= 1 \text{ hora} / 4 \text{ falhas} = \\ &0,25 \text{ hora/falha} = \\ &15 \text{ minutos para reparar falha (em média)} \end{aligned}$$

Exemplo: cálculo da disponibilidade

*“Um serviço deveria operar corretamente durante 9 horas.
Durante este período, ocorreram 4 falhas.
O tempo de todas as falhas totalizou 1 hora.”*

- Disponibilidade = $MTBF / (MTBF + MTTR)$

$$Disponibilidade = 2 / (2 + 0,25) = 88\% \text{ uptime}$$

Melhoria da Disponibilidade

- MTBF é uma medida básica da fiabilidade de um sistema
 - Queremos que o MTBF aumente
 - Por exemplo, através do aumento da qualidade do serviço
 - O sistema é mais fiável se o tempo entre falhas aumentar
- MTTR indica a eficiência da reparação
 - Queremos que o MTTR diminua
 - O sistema é mais fiável se o tempo de reparação diminuir
- Queremos disponibilidade a convergir para 100%
 - Mas sabemos que nunca lá vamos chegar...

Classes de Disponibilidade

Tipo	Indisponibilidade (min/ano)	Disponibilidade	Classe
Não gerido	52 560	90%	1
Gerido	5 256	99%	2
Bem gerido	526	99.9%	3
Tolerante a faltas	53	99.99%	4
Alta disponibilidade	5	99.999%	5
Muito alta disponibilidade	0.5	99.9999%	6
Ultra disponibilidade	0.05	99.99999%	7

D: Disponibilidade
(também chamado “número de **noves** de disponibilidade”)

Classe de Disponibilidade = $\log_{10} [1 / (1 - D)]$

Designação das classes de disponibilidade

Level of availability	Availability (percent)	Downtime per year
Commercial or standard	99.5	43.8 hours
Highly available	99.9	8.75 hours
Fault resilient	99.99	53 minutes
Fault tolerant	99.999	5 minutes
Continuous	100	0 minutes

Exemplos de sistemas e respetiva classe de disponibilidade

- Especificações existentes:
 - Classe 5: equipamento de monitorização de reatores nucleares
 - Classe 6: central telefónica
 - Classe 9: computador de voo

Algoritmos tolerantes a faltas

Tolerância a faltas

- Duas técnicas principais
 - Recuperação “para trás”
 - Recuperação “para a frente”

Recuperação “para trás”

- Guardar o estado algures, periodicamente ou em momentos relevantes para a aplicação
 - Usando um algoritmo para salvaguarda distribuída (estudaremos mais à frente)
- Quando um ou mais processos falham, estes são relançados o mais rapidamente possível
- Os novos processos lêem o último estado guardado e recomeçam a execução a partir desse estado
- Tipicamente designa-se esta solução por “checkpoint-recovery”

Recuperação para a frente

- São mantidas várias cópias (réplicas) do processo
- Quando uma réplica é alterada as restantes réplica devem também ser actualizadas
- Se uma réplica falhas os clientes podem usar outra réplica

Recapitulando...

- Na recuperação para trás:
 - Os estados guardados pelos vários **processos que falharam** devem estar mutuamente coerentes
- Na recuperação para a frente
 - Os estado das **réplicas que sobrevivem** devem estar mutuamente coerentes

Replicação tolerante a faltas

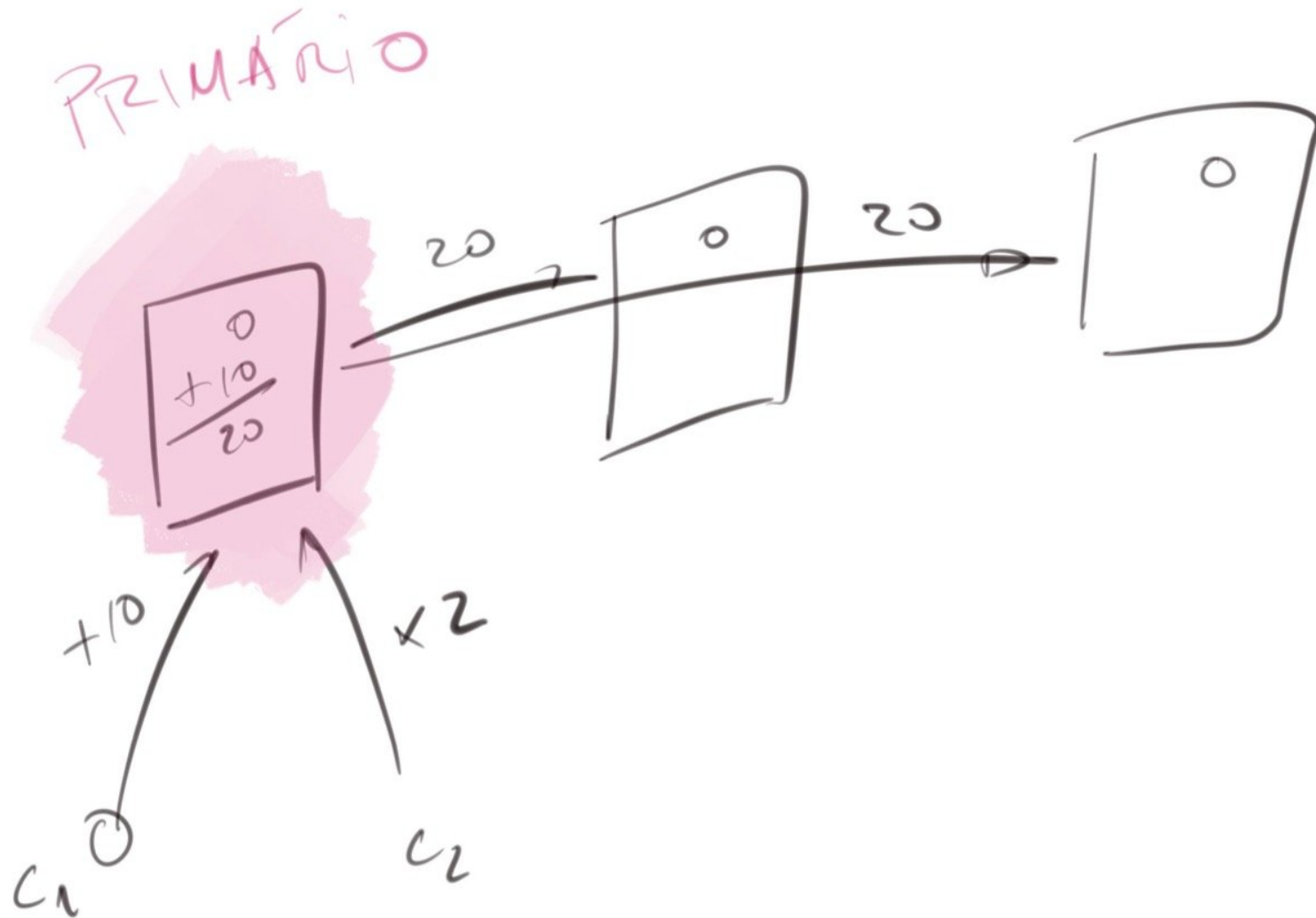
Tentemos construir algoritmos de replicação genéricos (*por recuperação para a frente*)

- Duas variantes principais
 - Primário-secundário
 - Replicação de máquina de estados
- Objetivos:
 - Assegurar **linearizabilidade** (coerência forte)
 - Tolerar faltas

Primário-secundário (um esboço)

- Um processo é eleito como primário
 - Se o primário falha, é eleito um novo primário
- Os clientes enviam pedidos ao primário
- Para cada pedido recebido, o primário:
 - Executa o pedido
 - Propaga o estado para os secundários e aguarda confirmação de todos
 - Responde ao cliente
 - (E processa o próximo pedido que esteja na fila de pedidos)

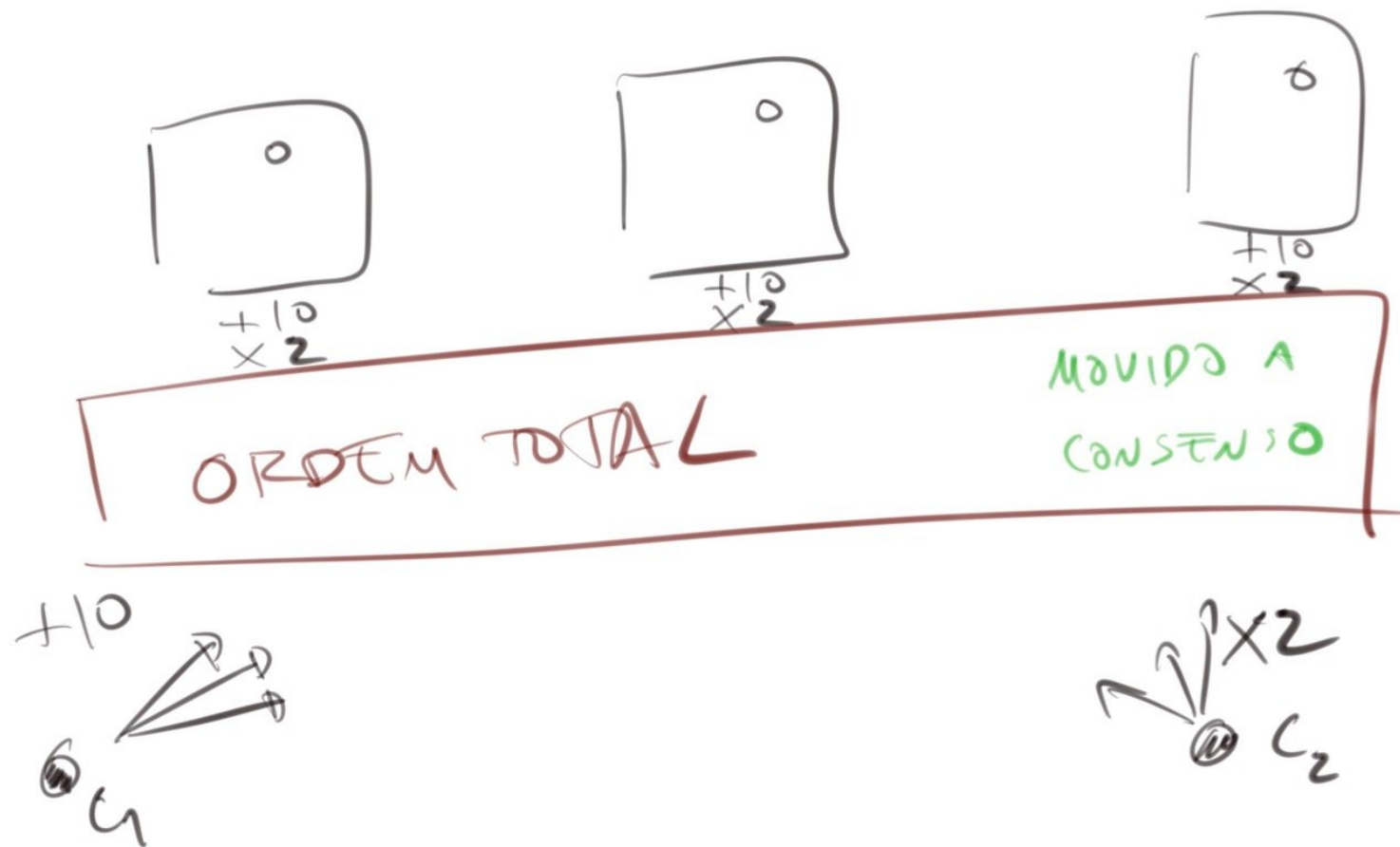
Primário-secundário (um esboço)



Replicação de máquina de estados (um esboço)

- Os clientes **enviam os pedidos para todas as réplicas**
- Todos os pedidos são **ordenados por ordem total**
- Todas as replicas processam os mesmos pedidos, pela mesma ordem
 - Assume-se que operações são determinísticas => replicas ficarão idênticas

Replicação de máquina de estados (um esboço)



Primário secundário vs RME

- Primário-secundário
 - Suporta operações não deterministas (o líder decide o resultado)
 - Se o líder produzir um valor errado, este valor é propagado para as réplicas
- Replicação máquina de estados
 - Se uma réplica produzir um valor errado não afecta as outras réplicas
 - As operações necessitam de ser deterministas

Abstrações para a construção de sistemas replicados

Abstracções para a construção de sistemas replicados

- Canais Perfeitos
- Difusão Fiável Regular
- Difusão Fiável Uniforme
- Difusão Atómica
- Sincronia na Vista

Canais Perfeitos

- Informalmente:
 - Garante a entrega de mensagens, ponto a ponto, de forma ordenada, no caso em que tanto o emissor como o destinatário não falham
- Como fazer:
 - Retransmitir uma mensagem até que a receção desta seja confirmada pelo destinatário
 - Usar ids de mensagens e não entregar uma mensagem antes daquelas com id inferior já terem sido entregues

Difusão Fiável

- Dois eventos:
 - `multicast(group, message)`
 - `deliver(message)`
- Informalmente:
 - Permite enviar uma mensagem em difusão com a garantia que todos os destinatários recebem a mensagem ou nenhum recebe
- Duas variantes
 - Regular
 - Uniforme

Difusão Fiável Regular

- Seja m uma mensagem enviada para um grupo de processos $\{p1, p2, \dots, pN\}$ por um membro desse grupo.
- *Validade*: se um processo correto p_i envia m então to mais cedo ou mais tarde p_i entrega m
- *Não-duplicação*: nenhuma mensagem m é entregue mais do que uma vez
- *Não-criação*: se uma mensagem m é entregue então m foi enviada por um processo correto
- *Acordo*: se um processo correto entrega m então todos os processos corretos entregam m

Tentemos implementar difusão fiável...

A straightforward way to implement *B-multicast* is to use a reliable one-to-one *send* operation, as follows:

To *B-multicast*(g, m): for each process $p \in g$, *send*(p, m);

On *receive*(m) at p : *B-deliver*(m) at p .

multicast, that a correct process will eventually deliver the message, as long as the multicaster does not crash. We call the primitive *B-multicast* and its corresponding basic

O que pode acontecer se o emissor puder falhar?

Difusão Fiável Regular: algoritmo

- O emissor envia a mensagem usando canais perfeitos para todos os membros do grupo
- Quando um membro do grupo recebe a mensagem, entrega-a à aplicação e reenvia-a para todos os membros do grupo

Limitações desta solução?...

Difusão Fiável Uniforme

- Seja m uma mensagem enviada para um grupo de processos $\{p1, p2, ..., pN\}$ por um membro desse grupo.
- *Validade*: se um processo correto p_i envia m então to mais cedo ou mais tarde p_i entrega m
- *Não-duplicação*: nenhuma mensagem m é entregue mais do que uma vez
- *Não-criação*: se uma mensagem m é entregue então m foi enviada por um processo correto
- *Acordo*: se um processo ~~correcto~~ entrega m então todos os processos corretos entregam m

Em que situações precisamos de difusão fiável uniforme (em vez de regular)?

Difusão Fiável Uniforme: algoritmo

Figure 15.9 Reliable multicast algorithm

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(g, m); // $p \in g$ is included as a destination

On B-deliver(m) at process q with g = group(m)

if (m \notin Received)

then

Received := Received \cup {m};

if (q \neq p) then B-multicast(g, m); end if

R-deliver m;

end if

Sincronia na Vista

- Informalmente:
 - Permite mudar a filiação de um grupo de processos de uma forma que facilita a tolerância a faltas

Sincronia na Vista

- Vista: conjunto de processos que pertence ao grupo
 - Novos membros podem ser adicionados dinamicamente
 - Um processo pode sair do grupo voluntariamente ou ser expulso case falhe
- O sistema evolui através de uma sequência totalmente ordenada de vistas
- Exemplo:
 - $V1 = \{p1, p2, p3\}$
 - $V2 = \{p1, p2, p3, p4\}$
 - $V3 = \{p1, p2, p3, p4, p5\}$
 - $V4 = \{p2, p3, p4, p5\}$

Sincronia na Vista

- Um processo considera-se correcto numa vista V_i se faz parte da vista V_i e faz parte da vista V_{i+1}
 - Por oposição, um processo que faz parte da vista V_i e que não faz parte da vista V_{i+1} , pode ter falhado durante a vista V_i
- Uma aplicação à qual já foi entregue a vista V_i mas à qual ainda não foi entregue a vista V_{i+1} diz-se que “está na vista V_i ”

Sincronia na Vista

- Uma aplicação que usa o modelo de sincronia na vista recebe vistas e mensagens
- Se uma aplicação envia uma mensagem m quando está na vista V_i , a mensagem m diz-se que foi enviada na vista V_i
- Se uma mensagem m é entregue à aplicação depois da vista V_i ser entregue e antes da vista V_{i+1} ser entregue, a mensagem m diz-se que foi entregue na vista V_i
- Uma mensagem m enviada na vista V_i é entregue na vista V_i

Sincronia na Vista

- Difusão fiável *síncrona na vista*:
 - Se um processo correto p na vista V_i envia uma mensagem m na vista V_i , então m é entregue a p na vista V_i
 - Se um processo entrega uma mensagem m na vista V_i , todos os processos correctos da vista V_i entregam m na vista V_i
- Corolário:
 - Dois processos que entregam a vista V_i e a vista V_{i+1} entregam exactamente o mesmo conjunto de mensagens na vista V_i

Mudança de vista

- Para mudar a vista é necessário obrigar as aplicações interromper temporariamente a transmissão de mensagens de forma a que o conjunto de mensagens a entregar na vista seja finito
- Para além disso, é necessário executar um algoritmo de coordenação para garantir que todos os processos correctos chegam a acordo sobre:
 - Qual a composição da próxima vista
 - Qual o conjunto de mensagens a entregar antes de mudar a vista

Não estudaremos estes algoritmos em SD

Usar estas abstrações para
concretizar os algoritmos que
esboçámos hoje

Primário-secundário (agora concretizado)

- Réplicas usam **sincronia na vista** para lidar com falha do primário:
 - Quando o primário p , algum tempo depois será entregue nova vista sem p
 - Quando nova vista é entregue e o anterior primário não consta nela, os restantes processos elegem o novo primário
 - Pode ser o primeiro membro da nova vista
 - Ou podemos usar outro algoritmo de eleição de líder
 - O novo primário anuncia o seu endereço num serviço de nomes (para que os clientes o descubram)
- Primário usa **difusão fiável uniforme *síncrona na vista*** para propagar novos estados aos secundários
 - Só responde ao cliente quando a uniformidade está garantida

Difusão Atômica

- Informalmente:
 - Fiável: se uma réplica recebe o pedido, todas as réplicas recebem o pedido
 - Ordem total: todas as réplicas recebem os pedidos pela mesma ordem

Relembrando o dois algoritmos para ordem total, agora considerando faltas

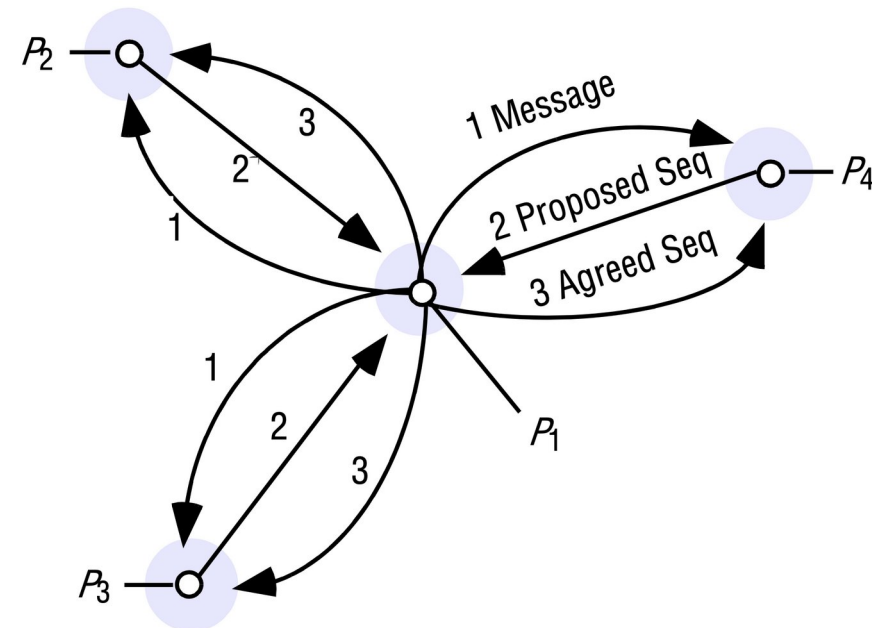
- Ordem total baseada em sequenciador
- Ordem total baseada em acordo colectivo

Ordem total baseada em sequenciador

- Quando existe uma falha, por exemplo do líder, as réplicas podem ser confrontadas com um estado incoerente:
 - Mensagens de dados que não foram ordenadas pelo líder
 - Mensagens do líder referentes a mensagens que ainda não chegaram
 - Para além disto, réplicas distintas podem ter perspetivas diferentes de qual o estado do sistema
- É necessário executar um algoritmo de recuperação complexo, baseado em consenso.

Acordo coletivo

- Quando há falhas sofre dos mesmos problemas e requer uma solução semelhante à referida anteriormente para a ordem total baseada num sequenciador



Replicação de máquina de estados (agora concretizado)

- Processos usam **sincronia na vista**
- Clientes usam **difusão atômica** *síncrona na vista* para enviar pedidos para todas as réplicas