Arquiteturas Tolerantes a faltas em Sistemas Distribuídos

Replicação

Replicação

- Conceito simples:
  - Manter cópias dos dados em múltiplos computadores

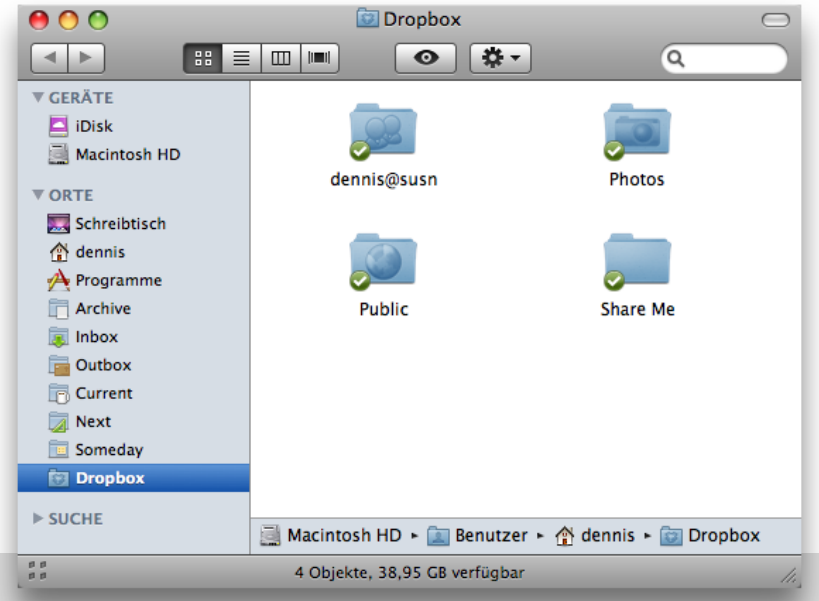
- Exemplos do nosso dia-a-dia?

Replicação: que benefícios?

- Melhor disponibilidade
  - O sistema mantém-se disponível mesmo quando:
    - Alguns nós falham
    - A rede falha, tornando alguns nós indisponíveis
- Melhor desempenho e escalabilidade
  - Clientes podem aceder às cópias mais próximas de si
    - Melhor desempenho
    - Caso extremo: cópia na própria máquina do cliente (*cache*)
  - Algumas operações podem ser executadas apenas sobre algumas das cópias
    - Distribui-se carga, logo consegue-se maior escalabilidade

Replicação: requisitos

- Transparência de replicação
  - Utilizador deve ter a ilusão de estar a aceder a um objeto lógico
  - Objeto lógico implementado sobre diferentes cópias físicas, mas sem que o utilizador se aperceba disso



Quantas falhas consegue um sistema replicado tolerar?

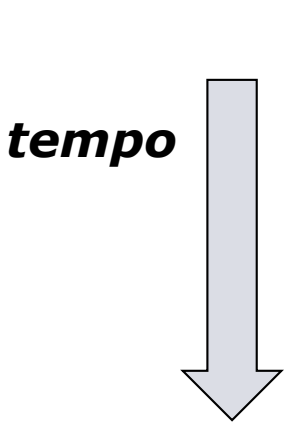
- Se as falhas forem silenciosas?
  - Esperaríamos que *f+1* réplicas tolerassem *f* nós em falha
  - Basta que uma réplica correcta responda para termos o valor correcto
- E se os nós puderem falhar de forma arbitrária (bizantina)?
  - Aí pode acontecer que, entre as respostas que recebermos, até a um máximo de f podem ser erradas
    - Logo a única alternativa é recebermos respostas iguais de pelo menos f+1 réplicas corretas
    - Logo esperaríamos que *2f+1* réplicas tolerassem *f* nós com falhas bizantinas
- Mas a realidade é mais complicada e normalmente precisamos de mais réplicas

Replicação: requisitos

- Consistência
  - Operações efetuadas sobre os objetos lógicos devem satisfazer a **especificação de correção** desses objetos

Consistência: exemplo

- Cópias de contas bancárias x e y mantidas em réplicas A e B
  - Clientes invocam operações sobre uma réplica
    - Mas tentam contatar outra réplica caso a primeira falhe
  - Operação é executada sobre a cópia local e retorna
  - Alteração é depois propagada à outra réplica



Client 1:	Client 2:
$setBalance_B(x, 1)$	
$setBalance_A(y, 2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

Problema?

Consistência

Como definir?

Antes de começarmos...

- Múltiplas cópias dos dados (réplicas), dois clientes
  - Cliente *i* invoca operações o_{i0}, o_{i1}, $o_{i2, \ldots}$
  - Operações síncronas: cliente espera pelo retorno antes de invocar próxima operação
- Cada réplica executa em série as operações de ambos os clientes
  - Exemplo: o_{10}, o_{11}, o_{20},o_{12}, o_{21}, o_{22},o_{13}, o_{23}, o_{14}
  - As réplicas não executam necessariamente as operações na mesma ordem

Linearizabilidade

Um sistema replicado diz-se linearizável sse (se e só se):

- Existe uma **serialização virtual** que:
  - É **correta** segundo a especificação dos objectos, e
  - Respeita o **tempo real** em que as operações foram invocadas
- A execução observada por cada cliente é consistente com essa serialização virtual (para todos os clientes)
  - Isto é, os valores retornados nas leituras são os mesmos

Captura uma execução possível caso o sistema não fosse replicado

Este exemplo é linearizável?

Client 1:	Client 2:
$setBalance\ (x, 1)$	
$setBalance\ (y, 2)$	
	$getBalance\ (y) \rightarrow 2$
	$getBalance\ (x) \rightarrow 0$

- Para responder, procurar uma serialização virtual que cumpra as condições anteriores
  - Correta
  - Respeita tempo real

E este outro exemplo, é linearizável?

Client 1:	Client 2:
$setBalance\ (x, 1)$	
	$getBalance\ (y) \rightarrow 0$
	$getBalance\ (x) \rightarrow 0$
$setBalance\ (y, 2)$	

Consistência sequencial

Um sistema replicado diz-se **sequencialmente consistente** sse:

- Existe uma **serialização virtual** que:
  - É **correta** segundo a especificação dos objetos, e
  - Respeita ~~o **tempo real**~~ a **ordem do programa de cada cliente**
- A execução observada por cada cliente é consistente com essa serialização virtual (para todos os clientes)

Condição mais fraca

Este exemplo é sequencialmente consistente?

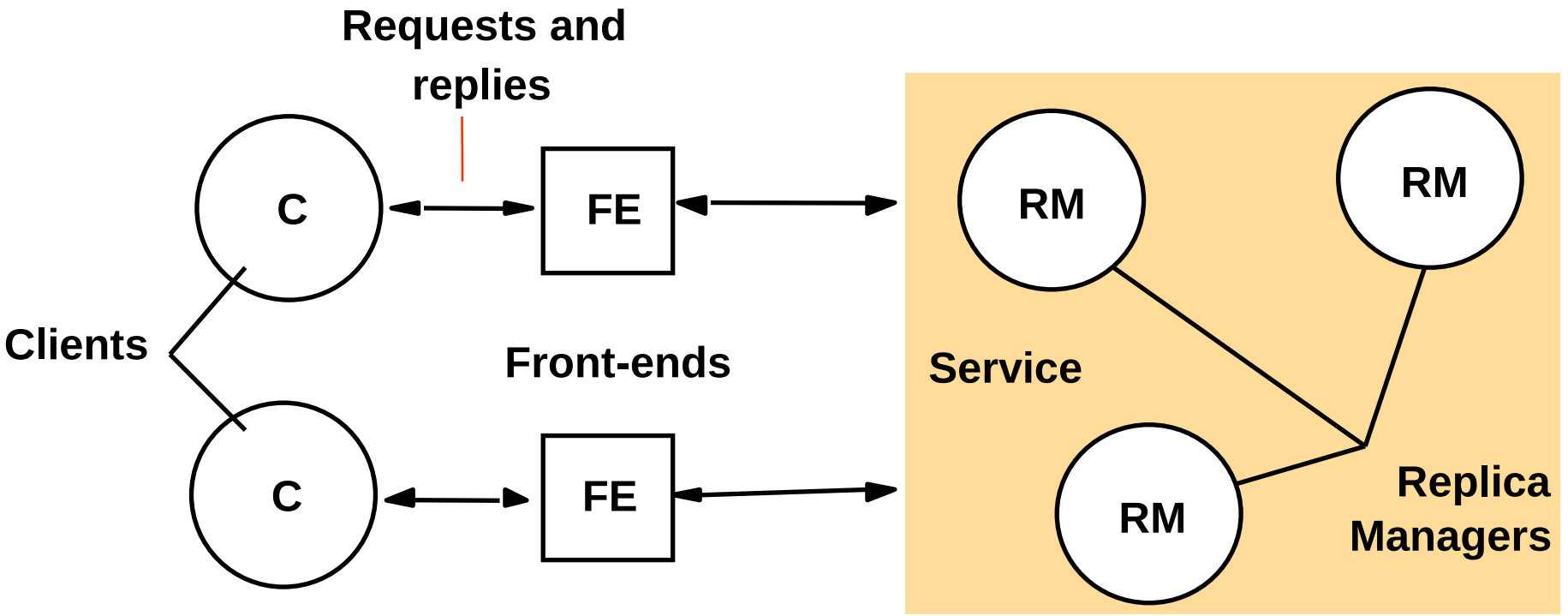
Client 1:	Client 2:
$setBalance\ (x, 1)$	
	$getBalance\ (y) \rightarrow 0$
	$getBalance\ (x) \rightarrow 0$
$setBalance\ (y, 2)$	

Consistência sequencial

- Condição mais fraca mas que permite implementações mais realistas do que a condição de linearizabilidade
- Para a maioria das aplicações,
  a consistência sequencial é suficiente
- Daqui para a frente tentaremos construir soluções que ofereçam esta garantia

Sistemas replicados

Modelo arquitetural de base



As cinco fases de uma invocação num sistema replicado

- Pedido
  - O *front-end* envia o pedido a um ou mais gestores de réplica
- Coordenação
  - Os gestores de réplicas coordenam-se para executarem o pedido consistentemente
- Execução
  - Cada gestor de réplica executa o pedido
- Acordo
  - Os gestores de réplicas acordam qual o efeito do pedido
- Resposta
  - Um ou mais gestores de réplica respondem ao cliente

Diferentes soluções podem omitir ou reordenar algumas fases.

Pressupostos nos slides seguintes

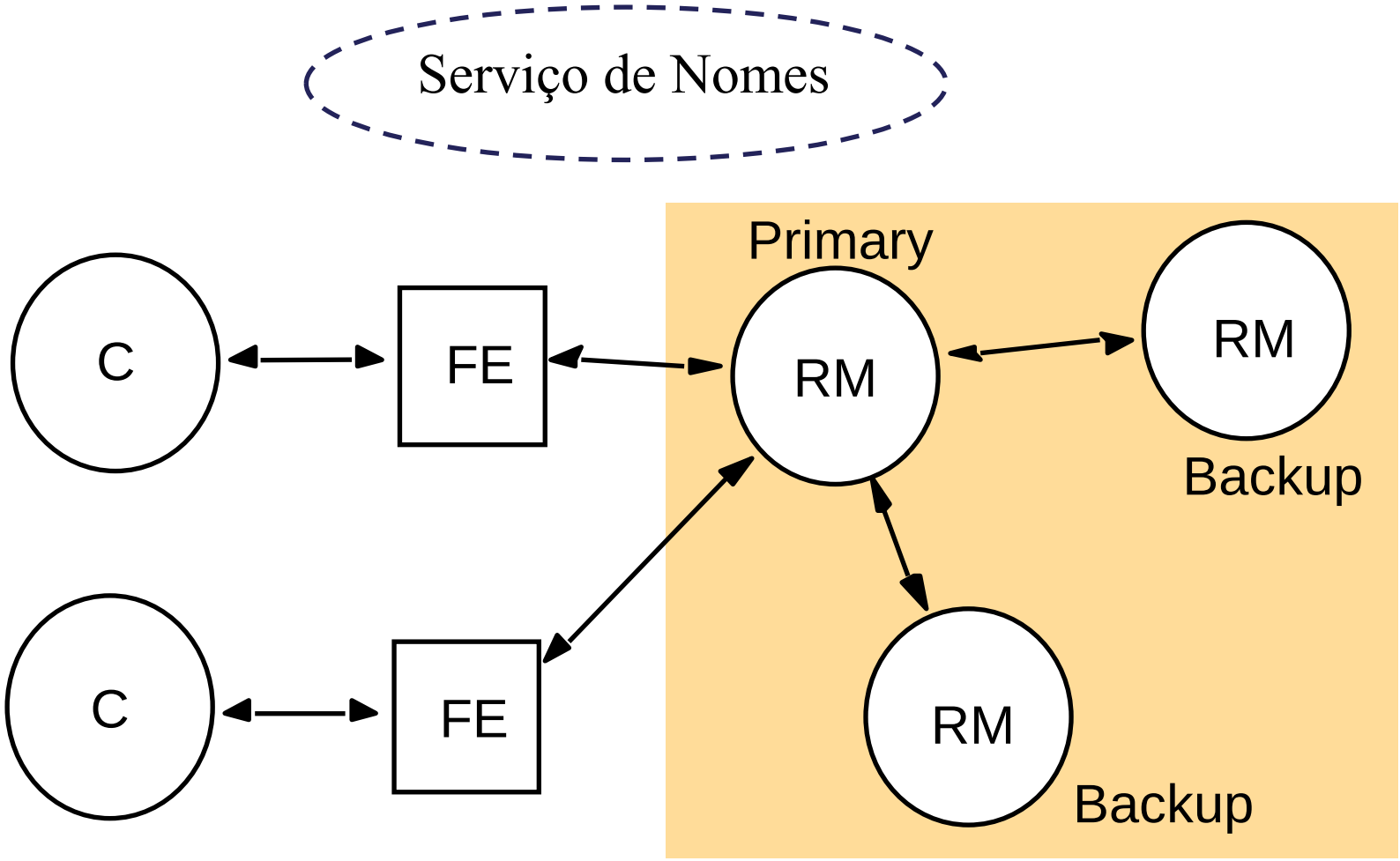
- Processos podem falhar silenciosamente
  - Ou seja, não há falhas arbitrárias de processos
- Operações executadas em cada gestor de réplica não deixam resultados incoerentes caso falhem a meio
- Replicação total
  - Cada gestor de réplica mantém cópia de todos os objetos lógicos
- Conjunto de gestores de réplica é estático e conhecido *a priori*

Replicação Passiva vs. Ativa

Replicação Passiva ***(primary-backup)***	Os clientes interatuam com um servidor principal. Os restantes servidores estão de reserva (*backups*), quando detetam que o servidor primário falhou, um deles torna-se o primário; Politica de Recuperação da falta
Replicação Ativa	Todos os servidores recebem pela mesma ordem os pedidos dos clientes, efetuam a operação, determinam qual o resultado correto por votação, e respondem ao cliente. Política de Compensação da falta

Replicação Passiva

Replicação passiva: arquitetura



Um Protocolo Simples de Replicação Passiva

Pedido
- FE envia pedido ao primário
  - Usando semântica no-máximo-1-vez

Coordenação
- Primário ordena os pedidos por ordem de chegada
  - Se pedido repetido, devolve resposta já guardada

Exec.
- Primário executa pedido e guarda resposta

Acordo
- Primário envia aos secundários: (novo estado, resposta, id.pedido)

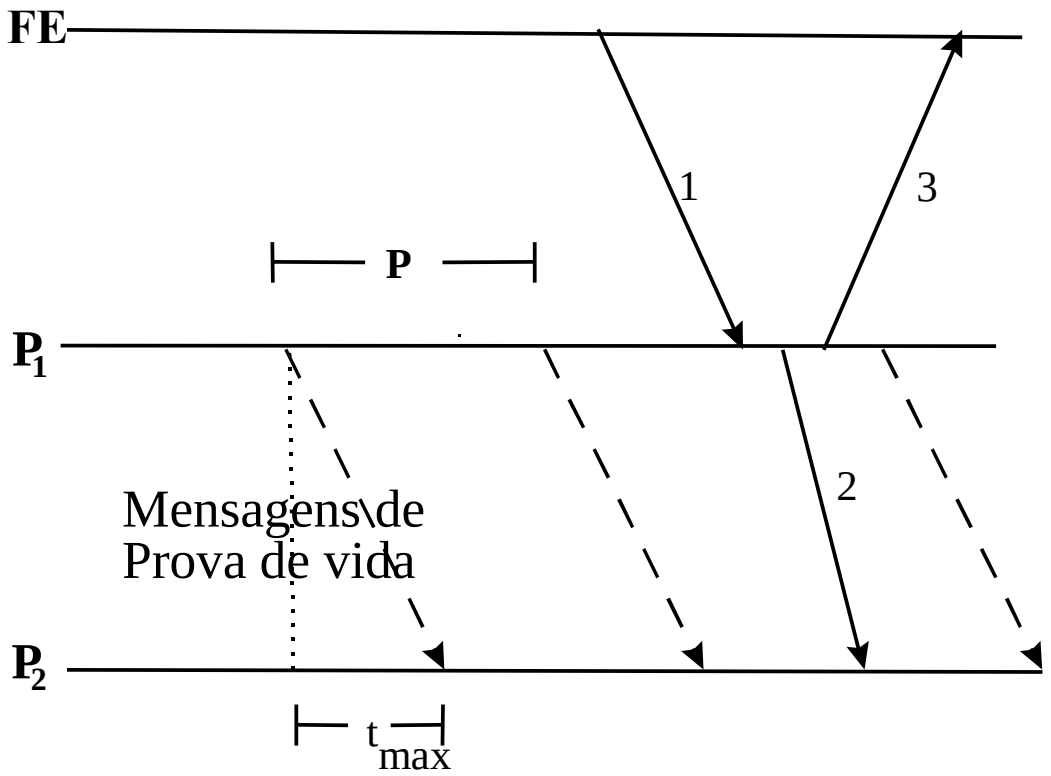
Resposta
- Primário responde ao *front-end*, que retorna ao cliente

Que simplificações são possíveis com operações determinísticas?

Deteção da falha do primário
Solução com 1 secundário apenas

- Primário envia mensagens "*I'm alive*" a cada P unidades de tempo
- Se o secundário não receber mensagem "*I'm alive*" após expirar um temporizador, torna-se o primário:
  - Avisa os *front-ends* e/ou o registo de nomes
  - Começa a processar os pedidos
- Se *front-end* fez pedido e não recebeu resposta, reenvia o pedido para o novo primário (semânticas no-máximo-uma-vez do RPC)

Exemplo sem falhas

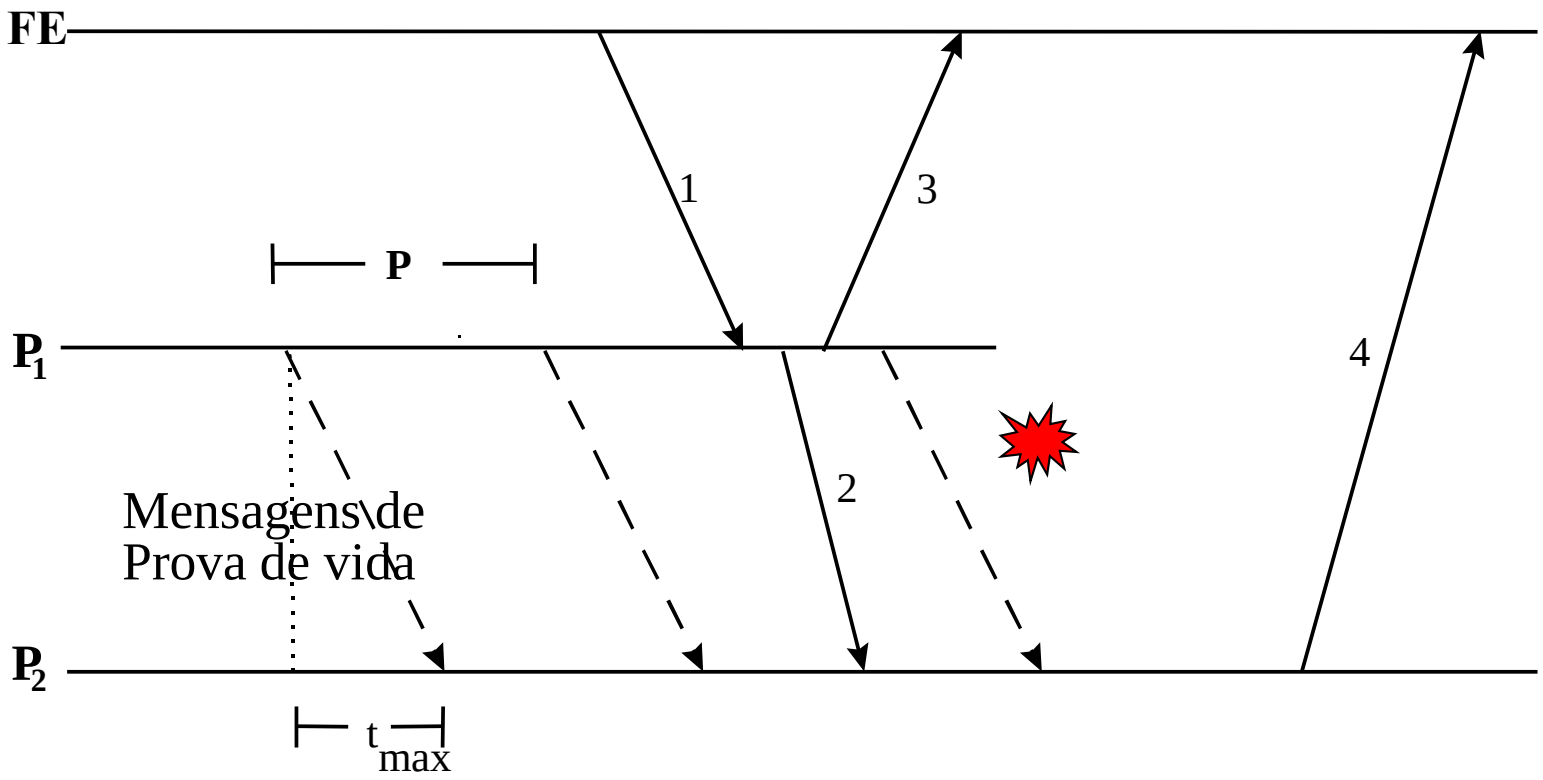


1 – Mensagem do cliente

2 – Mensagem de *update* do secundário

3 – Resposta ao cliente

Exemplo com falha do primário



1 – Mensagem do cliente

2 – Mensagem de *update* do secundário

3 – Resposta ao cliente

Em que instante P_2 pode assumir que é o primário?

Pressupostos

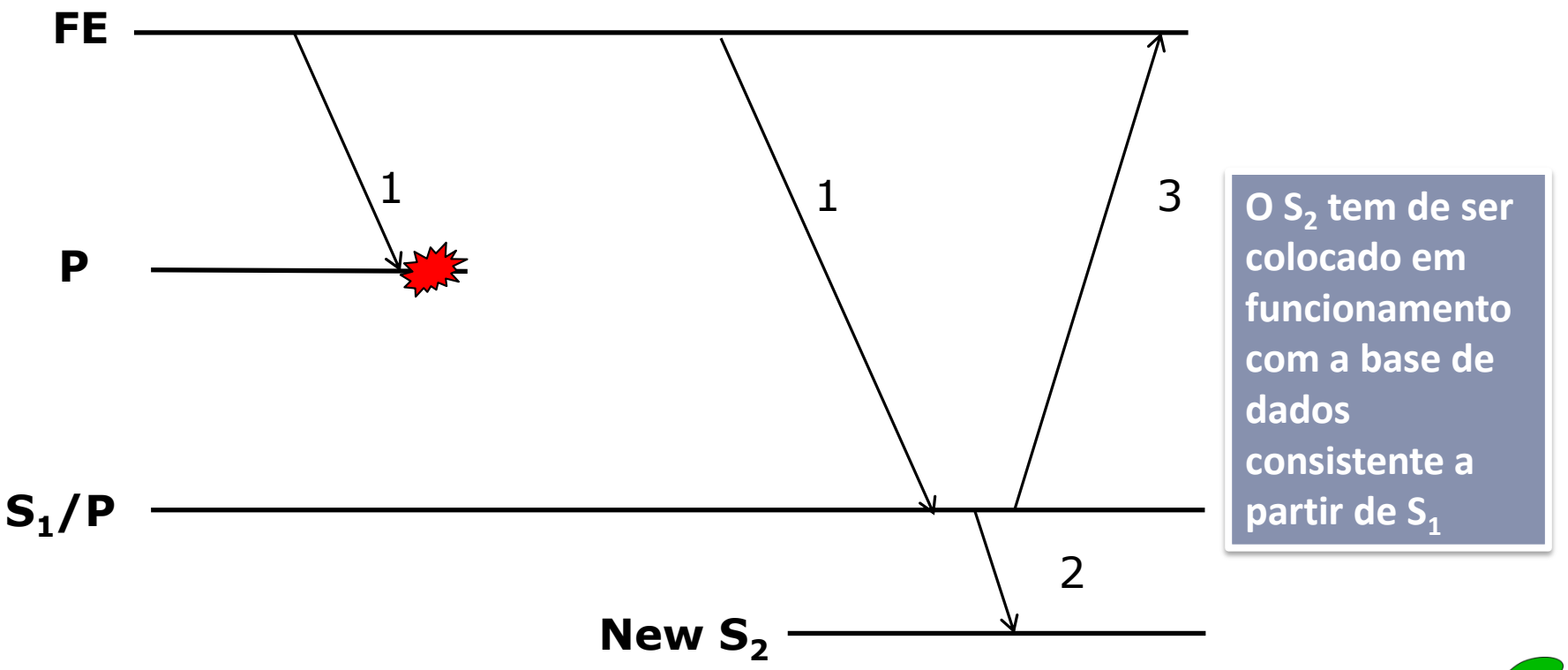
- A comunicação é fiável
  - O transporte recupera de faltas temporárias de comunicação e não há faltas permanentes
- Sistema síncrono; em particular, são conhecidos os limites máximos para:
  - Tempo de transmissão de uma mensagem na rede (tmax)
  - Tempo de processamento de pedido
  - Taxa de desvio dos relógios locais
- A rede assegura uma ordem FIFO na comunicação
- Os nós só têm faltas por paragem silenciosa
- A semântica de invocação dos RPC é no-máx-1-vez

O que pode acontecer se falharem estes pressupostos?

Solução garante consistência sequencial?

- Numa situação sem falhas do primário?
  - Sim, pois clientes interagem apenas com uma réplica (do primário)
- E caso o primário falhe e seja substituído por secundário?
  - Mais complicado de analisar

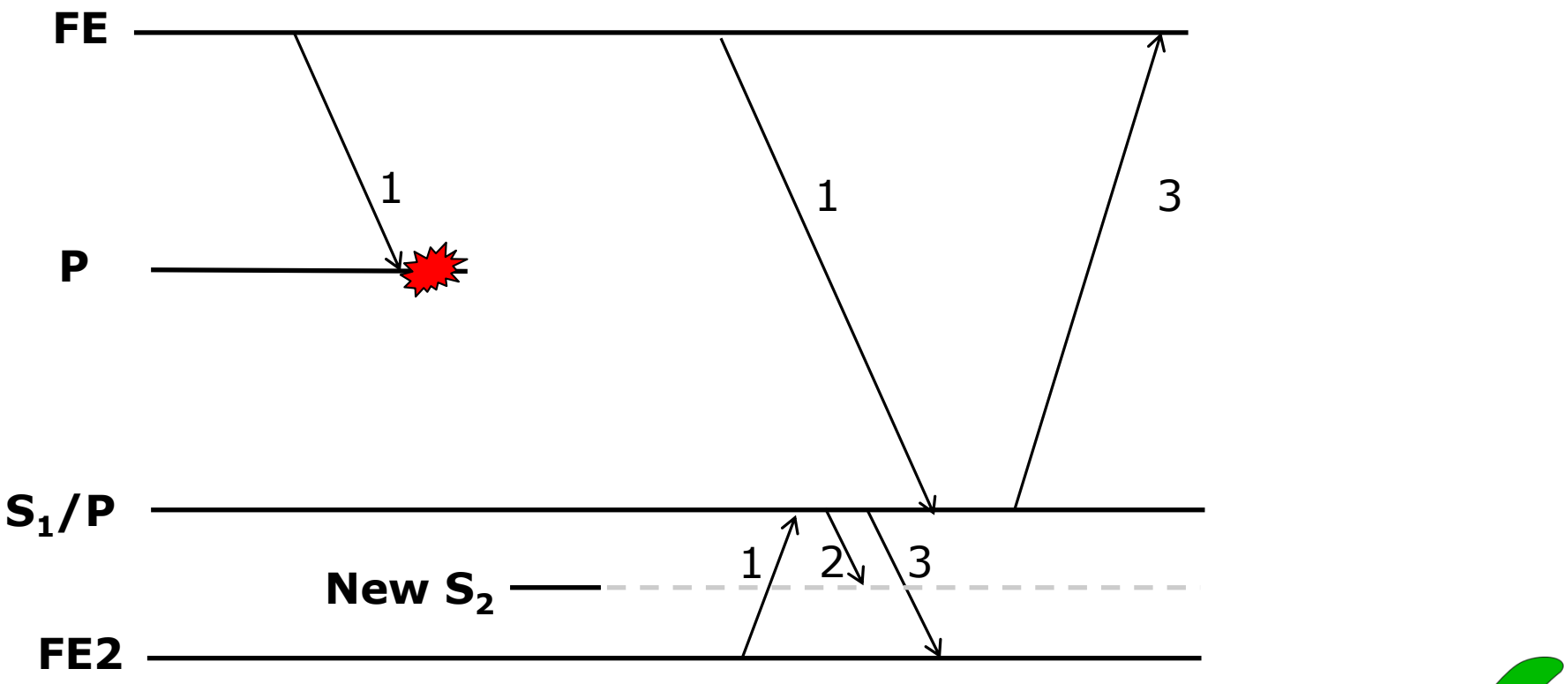
Cenários de Falha do primário (I)



O S_2 tem de ser colocado em funcionamento com a base de dados consistente a partir de S_1

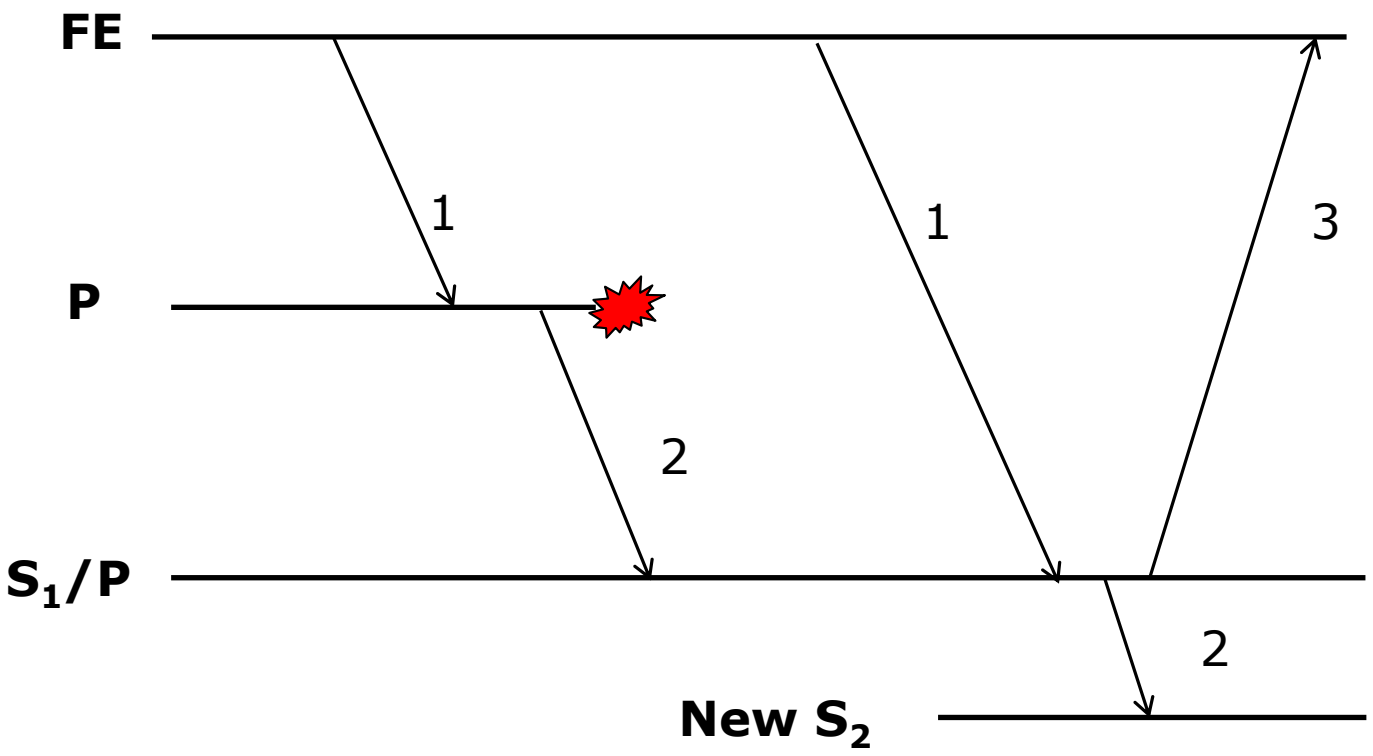
Sequencialmente consistente

Cenários de Falha do primário (II)



Sequencialmente consistente

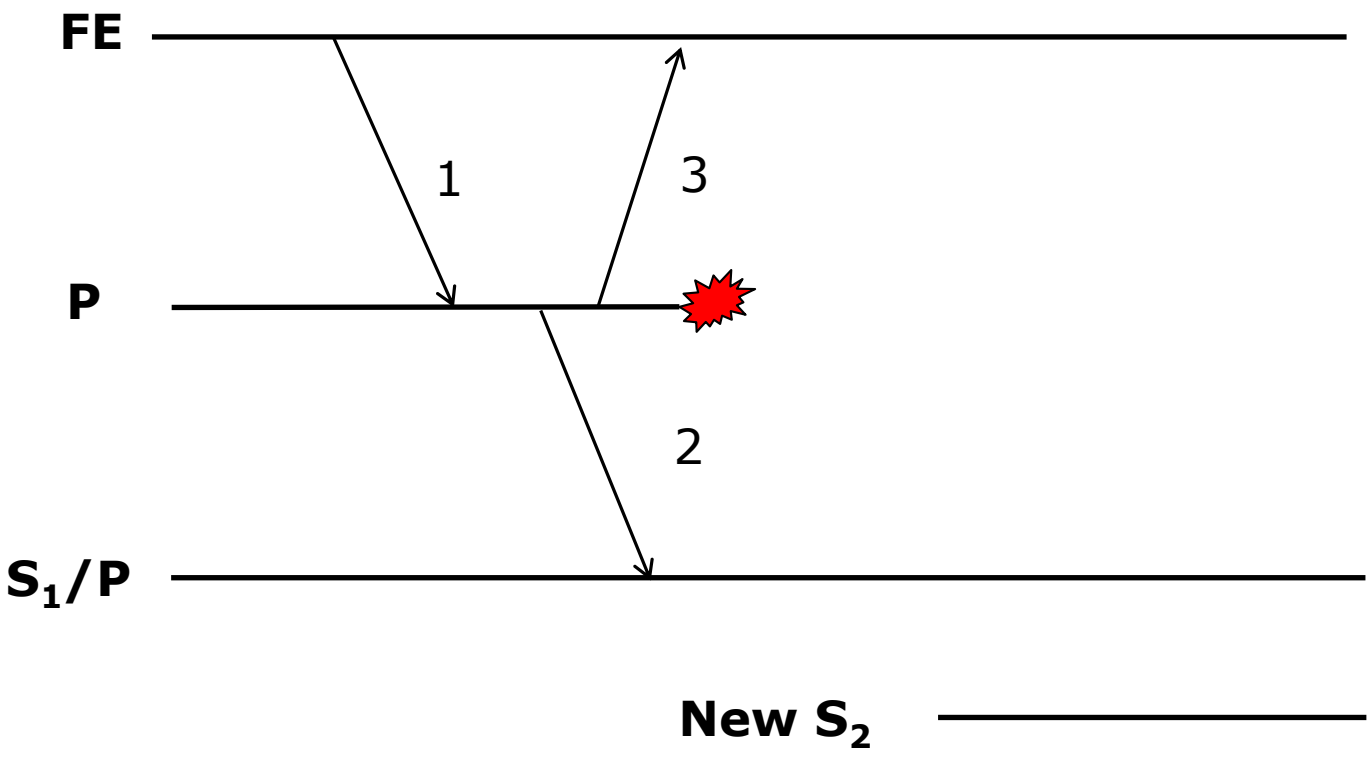
Cenários de Falha do primário (III)



Necessário que o FE reenvie o pedido com semântica no-máx-1-vez e receberá a resposta do secundário, que foi atualizado correctamente

Sequencialmente consistente

Cenários de Falha do primário (IV)



O sistema ficou consistente
O novo S tem de ser colocado em funcionamento com a base de dados consistente a partir de S1

Sequencialmente consistente

Probabilidades

P(A): Probabilidade de A (e.g., falta) acontecer numa dada unidade de tempo (P(A) << 1)

A, B, C: Acontecimentos independentes, sem memória

- $P(A \wedge B) = P(A) * P(B)$
- $P(A \vee B) = P(A) + P(B) - P(A)*P(B) \cong P(A) + P(B)$

Tempo médio até ao acontecimento (*Mean Time to Event*)

- $MT(A) = 1 / P(A)$

Tempo médio até um de vários acontecimentos A, B, C

- $MT(G) \cong 1 / [P(A) + P(B) + P(C)]$
  - $= 1 / [1/MT(A) + 1/MT(B) + 1/MT(C)]$

Tempo médio num sistema constituído por N sistemas do tipo

- $MT(NG) \cong MT(A) / N$

Evolução do MTBF na replicação

Probabilidade de falha num ano	MTBF (ano)	MTBF (2 servidores iguais sem qualquer protocolo de replicação)	MTBF (2 servidores em primary-backup)
0,5	2	1,333 (aprox 1)	4
0,1	10	5,26 (aprox 5)	100

Se a probabilidade de um servidor falhar for P

P(A ou B) = P(A) + P(B) - P(A)*P(B) aprox. P(A) + P(B)
P(A e B) = P(A) * P(B)

(Admitindo que as faltas são independentes boa aproximação no hardware mas inválida para faltas no software se for idêntico)

E se houver múltiplos secundários simultâneos?

- Após deteção da falha do primário, secundários disponíveis elegem o novo primário
  - Mais complicado
  - Necessário assegurar que todos os secundários elegem o mesmo primário
  - Matéria fora do âmbito das teóricas de SD

Como medir os custos da replicação?

- Grau de replicação:
  - Número de servidores usados para implementar o serviço
- Tempo de resposta (*blocking time*):
  - Tempo máximo entre um pedido e a sua resposta, no período sem falhas
- Tempo de recuperação (*failover time*):
  - Tempo desde falha do primário até cliente ser notificado do novo primário
- Objetivo: assumindo que f componentes podem falhar, minimizar as métricas acima

Custos da nossa solução

- Custos
  - Grau de replicação: **ótimo**
    - **f+1** réplicas toleram f faltas
  - Tempo de resposta: **2*t_{max}**
    - Ignorando tempo de processamento
  - Tempo de recuperação: P+3*t_{max}
    - Desde falha até cliente ser notificado
    - Assumindo situação mínima em que o secundário avisa o cliente; com um servidor de nomes é mais demorado
    - Assumindo que taxa de desvio dos relógios é nula