

SISTEMAS DISTRIBUÍDOS

Capítulo 8

Introdução à replicação e consistência

NOTA PRÉVIA

A apresentação utiliza algumas das figuras livro de base do curso

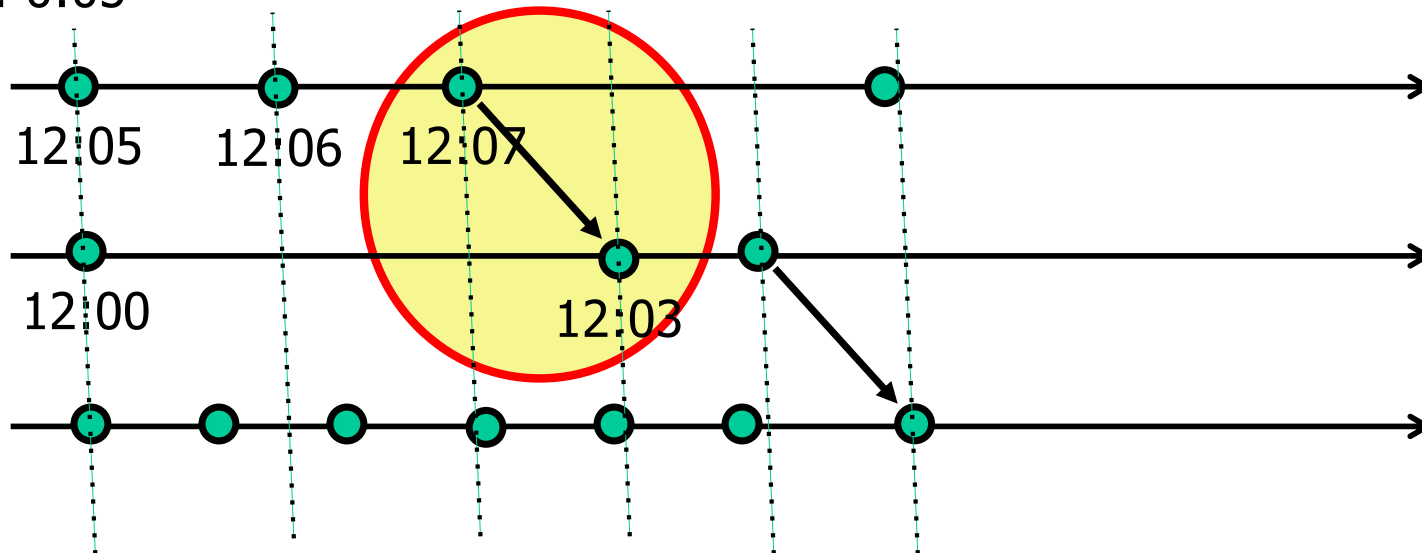
G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition

NA ÚLTIMA AULA....

Num sistema distribuído é impossível sincronizar os relógios de vários computadores a menos dum dado valor.

Assim, é impossível usar o valor do relógio em diferentes computadores para saber a ordem dos eventos.

eg.: erro: 0:05

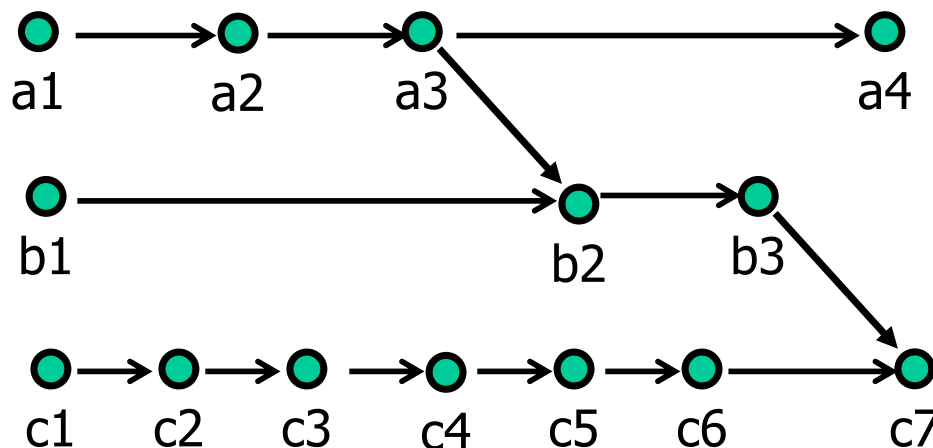


RELAÇÃO "ACONTECEU ANTES" (LAMPORT 1978)

e_1 aconteceu antes de e_2 , sse:

- e_1 e e_2 ocorreram no mesmo processo e e_1 ocorreu antes de e_2
- e_1 e e_2 são, respetivamente, os eventos de enviar e receber a mensagem m
- $\exists e_x: e_1 \rightarrow e_x$ e $e_x \rightarrow e_2$ (relação transitiva)

Dois eventos e_1, e_2 dizem-se concorrentes ($e_1 \parallel e_2$) se $\neg e_1 \rightarrow e_2$ e $\neg e_2 \rightarrow e_1$



"ACONTECEU ANTES": MECANISMOS

Dados dois eventos, e_1 e e_2 , $e_1 \rightarrow e_2 \Rightarrow C_{\text{lock}}(e_1) < C_{\text{lock}}(e_2)$

- ***Relógios lógicos*** (também chamado ***relógio de Lamport***): inteiro

Dados dois eventos, e_1 e e_2 , $C_{\text{lock}}(e_1) < C_{\text{lock}}(e_2) \Rightarrow e_1 \rightarrow e_2$

- ***História causal***: conjunto de (identificadores de) eventos
- ***Relógio vetorial***: array (ou mapa) de inteiros, que são sumário da história causal
- ***Vetor versão***: relógio vetorial, mas registando apenas eventos importantes

NESTE CAPÍTULO DA MATÉRIA....

Introdução à replicação

- Consistência forte. E.g.: primário-secundário
- Consistência fraca.

Caching

- Sistemas de ficheiros distribuídos
- Caching NFS
- Caching CIFS
- Caching Callback Promise

NESTE CAPÍTULO DA MATÉRIA....

Introdução à replicação

- **Consistência forte. E.g.: primário-secundário**
- Consistência fraca.

Caching

- Sistemas de ficheiros distribuídos
- Caching NFS
- Caching CIFS
- Caching Callback Promise

PROBLEMA

As falhas dos componentes são inevitáveis...

Como melhorar a disponibilidade de um serviço na presença de falhas dos componentes?

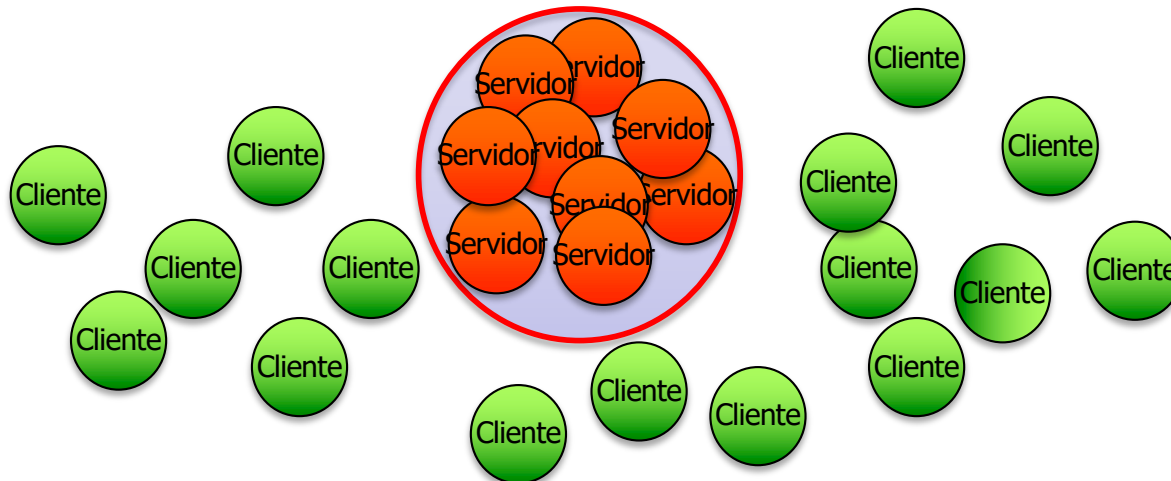
COMPONENTES REPLICADOS

Havendo componentes redundantes para o mesmo serviço, é possível continuar a fornecer o serviço enquanto existirem suficientes réplicas a operar.

VARIANTES DO MODELO CLIENTE/SERVIDOR: SERVIDOR

Cliente/servidor replicado

- Existem vários servidores **idênticos** (i.e. capazes de responder aos mesmos pedidos, de igual forma)



REPLICAÇÃO

Servidores/componentes idênticos...

- Capazes de responder aos mesmos pedidos, **de igual forma**

Igual código mas, sobretudo, **estado equivalente...**

OBJETIVO

Garantir que o estado de vários servidores é mantido equivalente...

... na presença de vários clientes a fazer modificações (e leituras) dos seus estados....

Como fazer?

PRIMEIRA ABORDAGEM: CONSISTÊNCIA FORTE

Objetivo: ter várias réplicas, mas que se comportem como se existisse apenas uma réplica que tolerasse falhas.

Modelo de falhas:

- as mensagens podem-se perder e trocar de ordem;
- um servidor pode falhar por crash, i.e., deixar de responder (e eventualmente recuperar).

Desafios:

1. Garantir que todas as réplicas convergem para o mesmo estado
2. Garantir que os clientes observam sempre a última escrita

PRIMEIRA ABORDAGEM: CONSISTÊNCIA FORTE

Objetivo: ter várias réplicas, mas que se comportem como se existisse apenas uma réplica que tolerasse falhas.

Modelo de falhas:

- as mensagens podem-se perder e trocar de ordem;
- um servidor pode falhar por crash, i.e., deixar de responder (e eventualmente recuperar).

Desafios:

1. **Garantir que todas as réplicas convergem para o mesmo estado**
2. Garantir que os clientes observam sempre a última escrita

REPLICAÇÃO DE MÁQUINA DE ESTADOS

Como garantir que todas as réplicas convergem para o mesmo estado ?

1. Ter operações deterministas
 - Em todas as execuções, executar f no estado S_a origina o estado S_d , i.e., $f(S_a) = S_d$
2. Executar a mesma sequência de operações em todas as réplicas
 - A partir do mesmo estado inicial S_0 , todas as réplicas chegam ao mesmo estado $\mathbf{f_n(...f_2(f_1(S_0)))}$ ao executarem as operações deterministas, f_1, f_2, \dots, f_n

REPLICAÇÃO DE MÁQUINA DE ESTADOS

Protocolo **primário/secundário** (passivo) – $2.f+1$ réplicas para tolerar **f** falhas

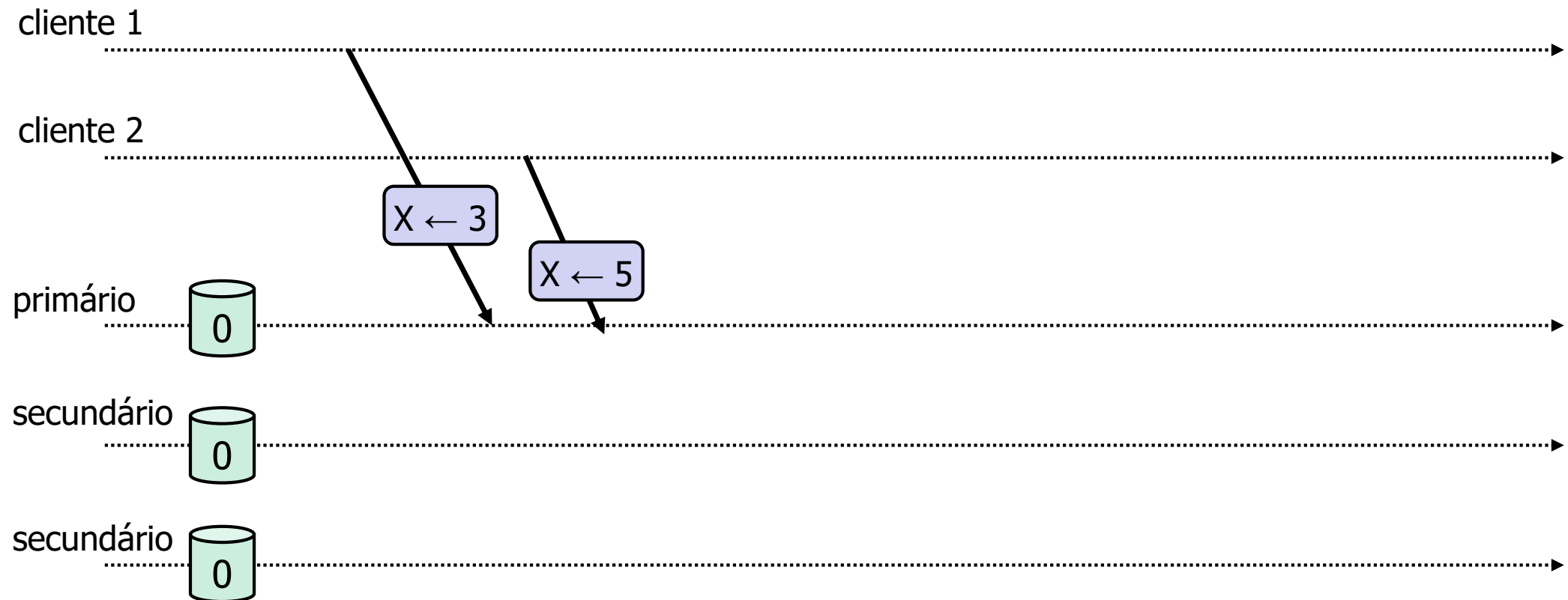
Primário: mantém a versão oficial dos dados

Secundário: mantém uma cópia da versão do primário



PRIMÁRIO / SECUNDÁRIO : ESCRITA

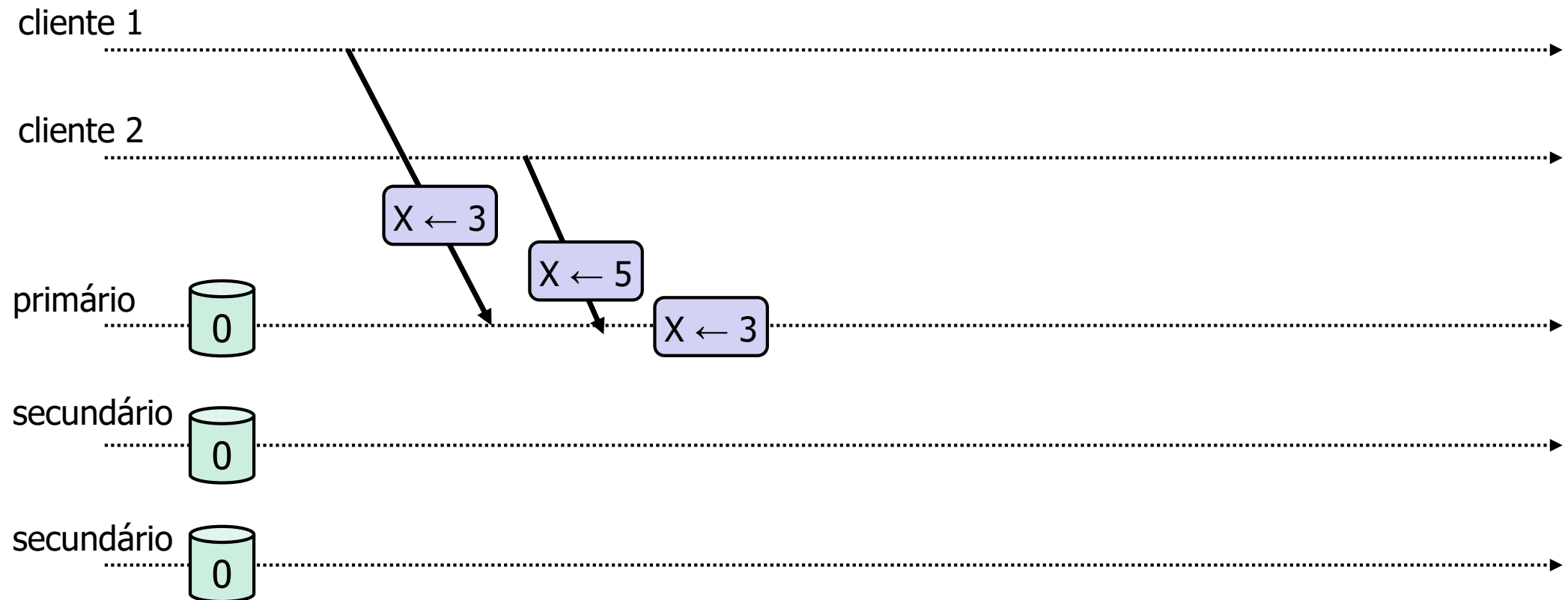
Cliente envia escrita para o primário



PRIMÁRIO / SECUNDÁRIO : ESCRITA

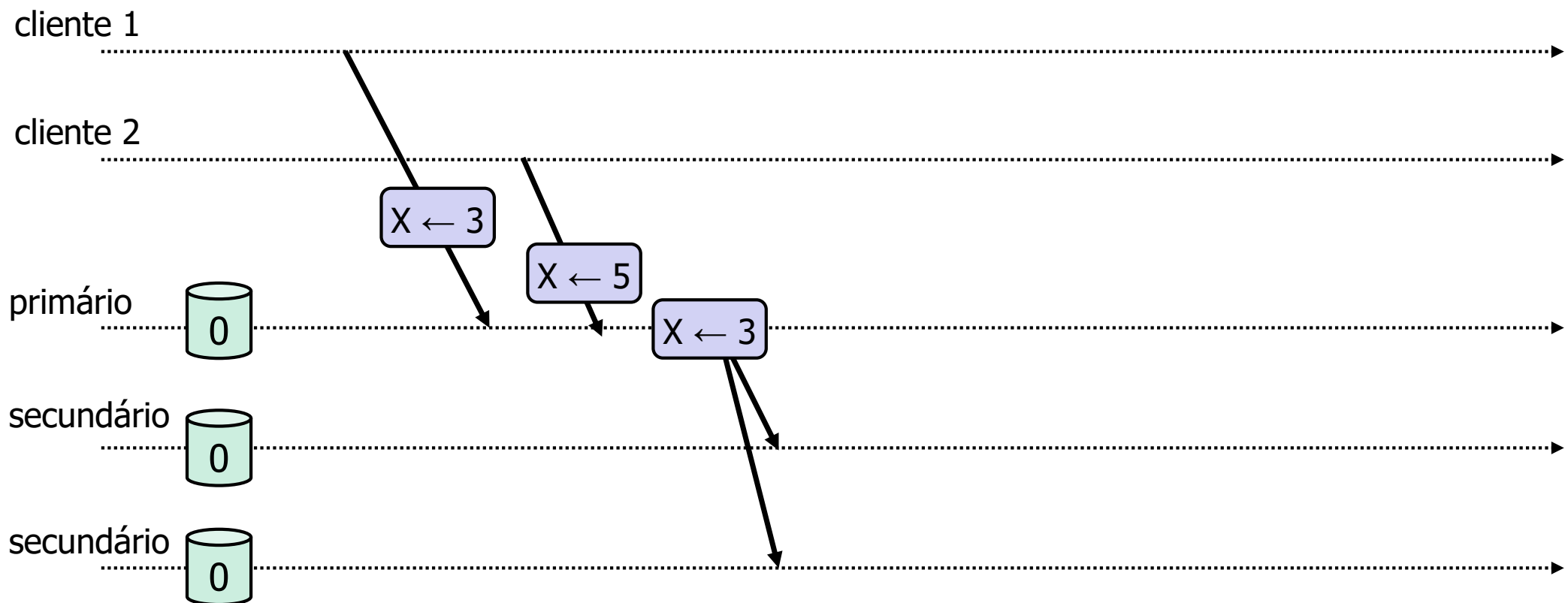
Cliente envia escrita para o primário

Primário **serializa as escritas** e executa-as ordenadamente



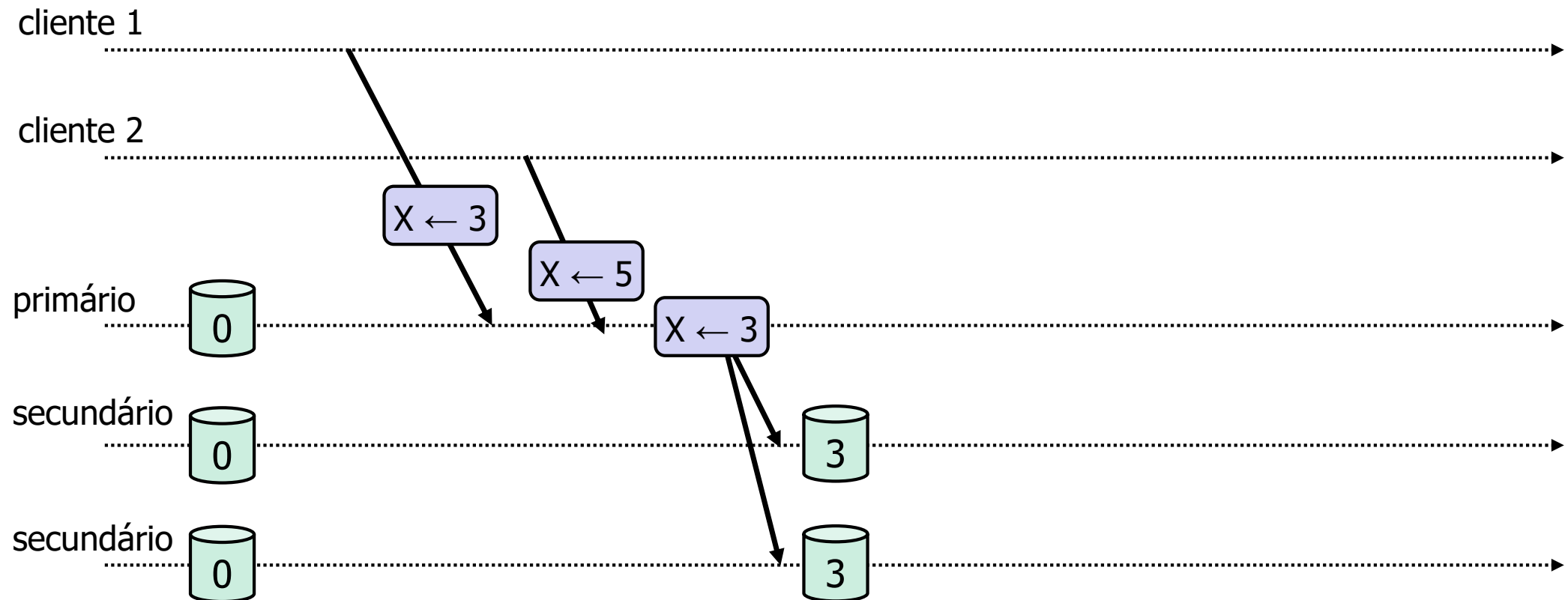
PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

1. Envio da operação para os secundários



PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

1. Envio da operação para os secundários
2. Secundário executa operação se $f=1$

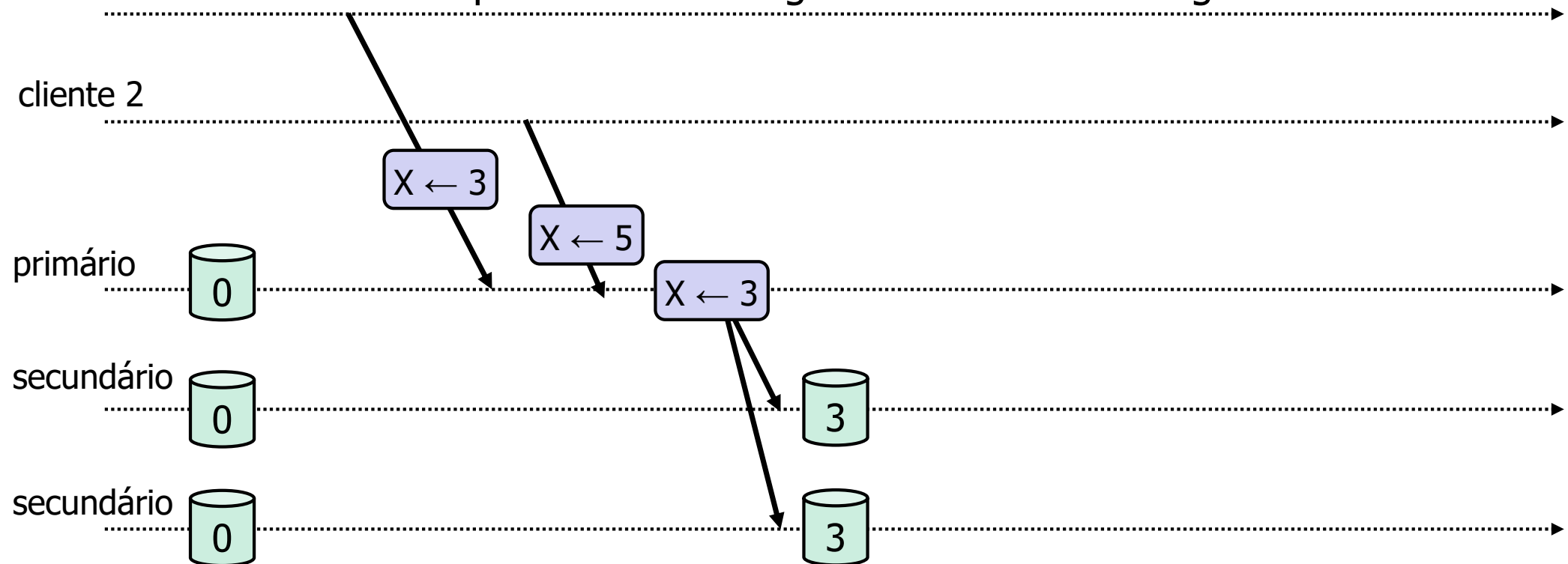


PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

1. Envio da operação para os secundários

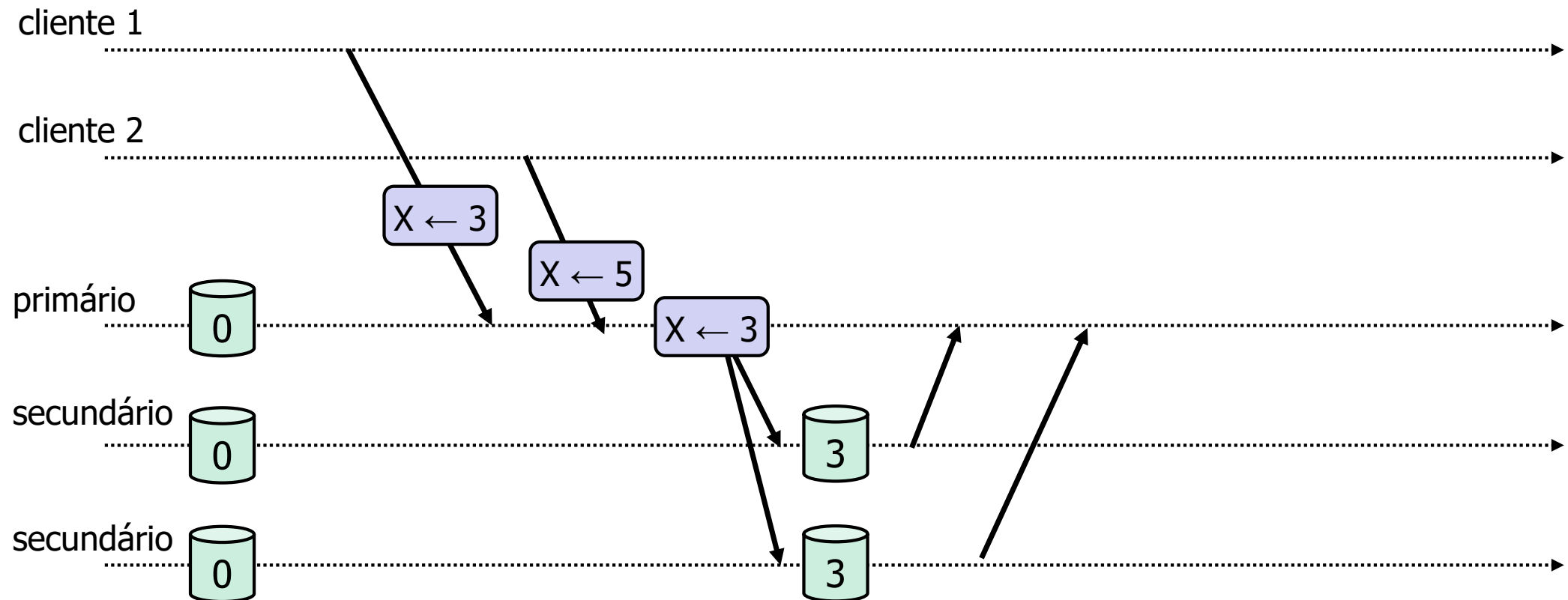
2. Secundário executa operação se $f=1$

NOTA: Com $f > 1$, um secundário não pode executar imediatamente a operação, porque esta pode-se perder se o primário e esse secundário cliente 1 forem os únicos que têm a mensagem e falharem de seguida.



PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

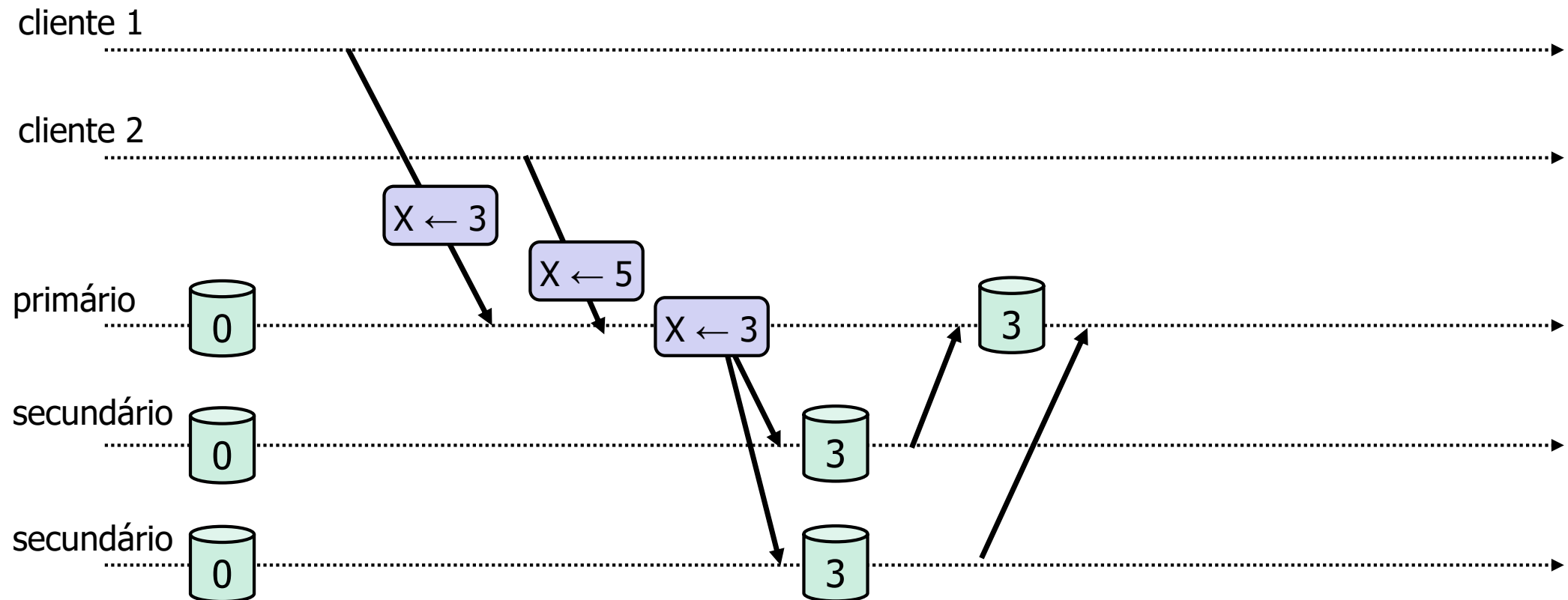
1. Envio da operação para os secundários
2. Secundário executa operação se $f=1$
3. Secundário envia ACK para primário



PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

Quando recebe f acks (com f o número de falhas a tolerar)

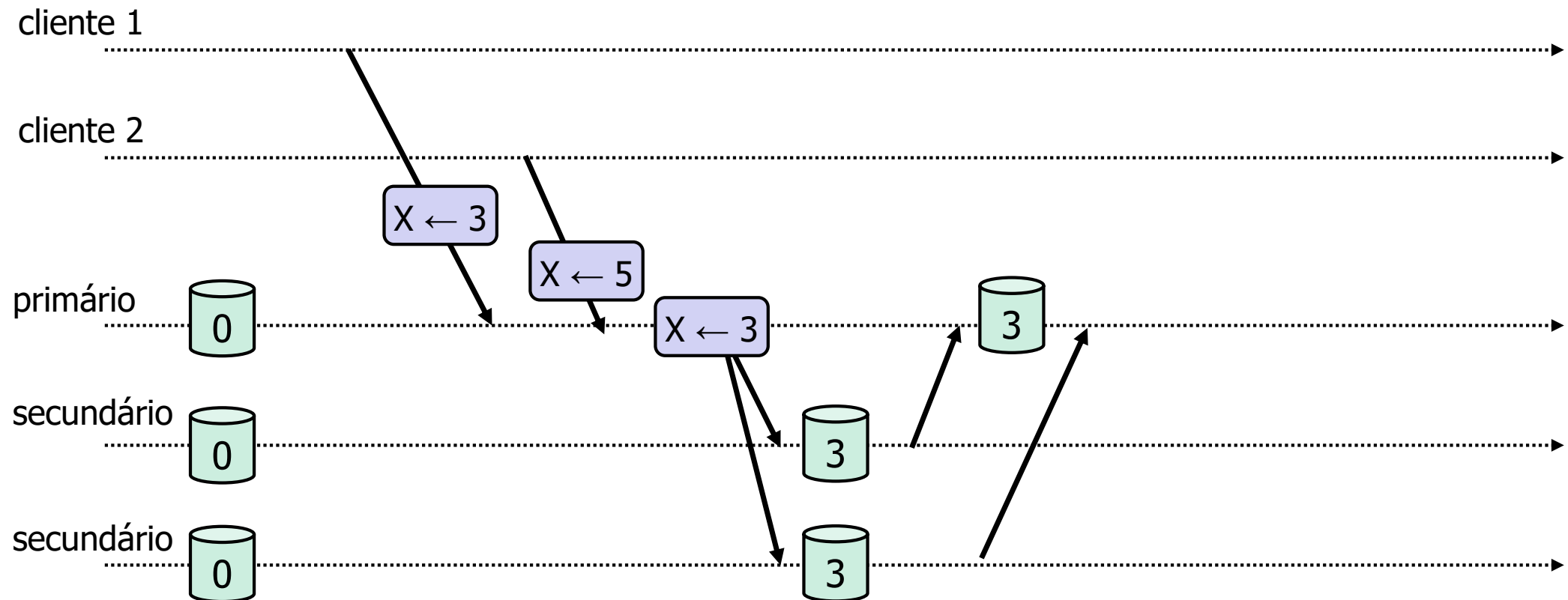
4. Execução no primário



PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

Quando recebe f acks (com f o número de falhas a tolerar)

4. Execução no primário

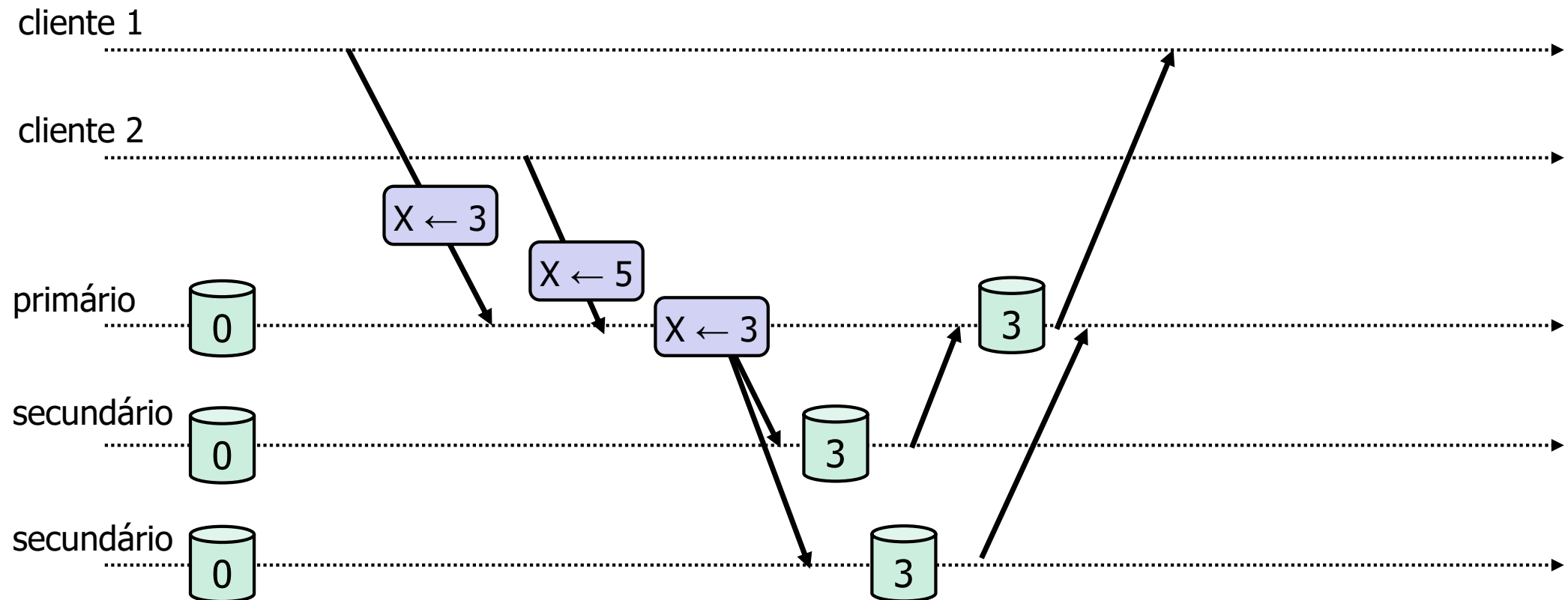


PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

Quando recebe f acks (com f o número de falhas a tolerar)

4. Execução no primário

5. Primário envia ACK para cliente

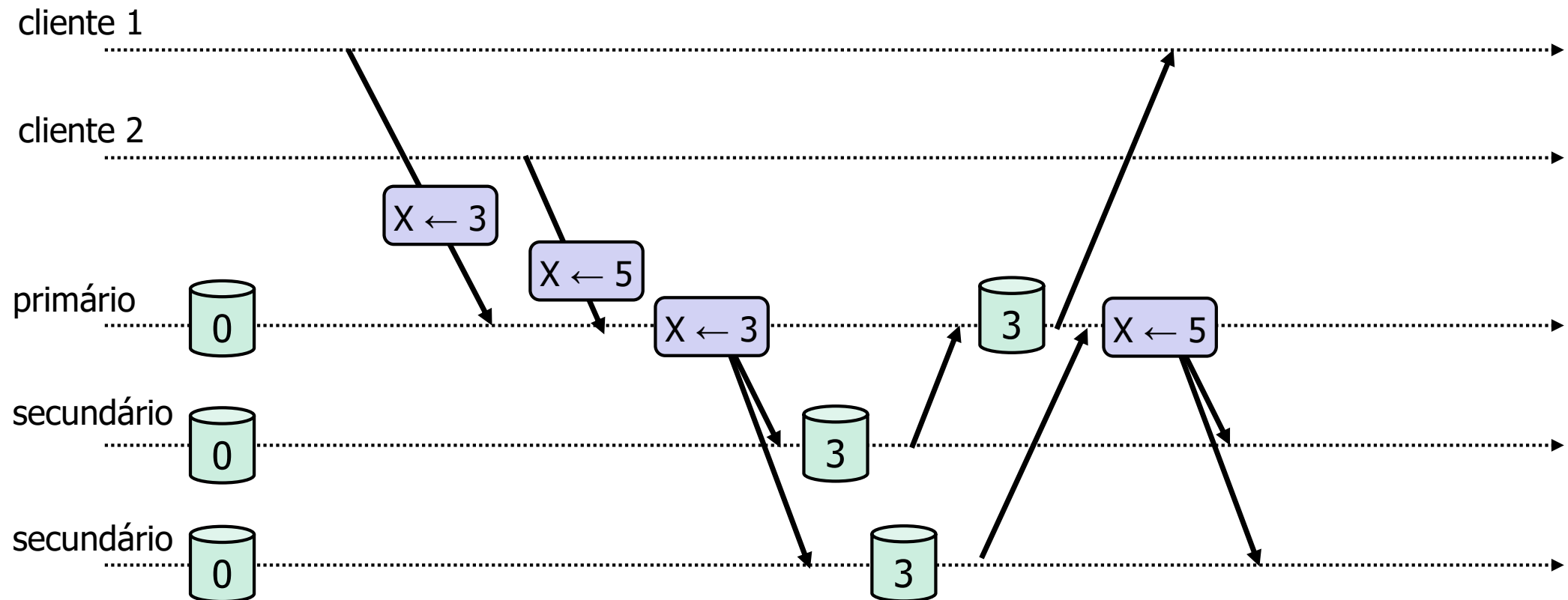


PRIMÁRIO / SECUNDÁRIO : ESCRITA : EXECUÇÃO

Quando recebe f acks (com f o número de falhas a tolerar)

4. Execução no primário

5. Primário envia ACK para cliente



PRIMEIRA ABORDAGEM: CONSISTÊNCIA FORTE

Objetivo: ter várias réplicas, mas que se comportem como se existisse apenas uma réplica que tolerasse falhas

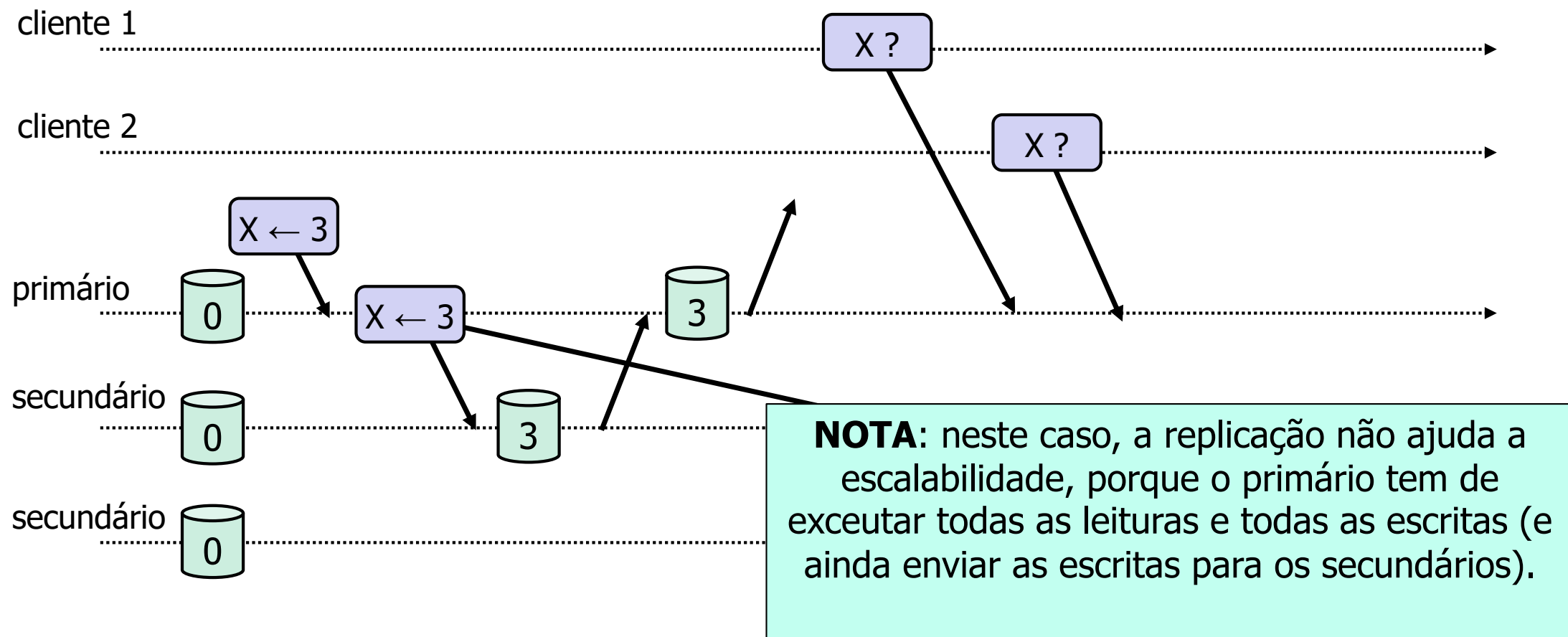
Desafios:

1. Garantir que todas as réplicas convergem para o mesmo estado
2. **Garantir que os clientes observam sempre a última escrita**

PRIMÁRIO / SECUNDÁRIO : LEITURA

Para se garantir que se observa a última escrita, os clientes têm de fazer as leituras no primário. **Porquê?**

Porque um secundário pode não ter recebido a última escrita, se a propagação do primário para o secundário falhou.



PRIMEIRA ABORDAGEM: CONSISTÊNCIA FORTE

Objetivo: ter várias réplicas, mas que se comportem como se existisse apenas uma réplica que tolerasse falhas

Desafios:

1. Garantir que todas as réplicas convergem para o mesmo estado
2. **Garantir que os clientes observam sempre a última escrita**

PRIMEIRA ABORDAGEM: CONSISTÊNCIA FORTE

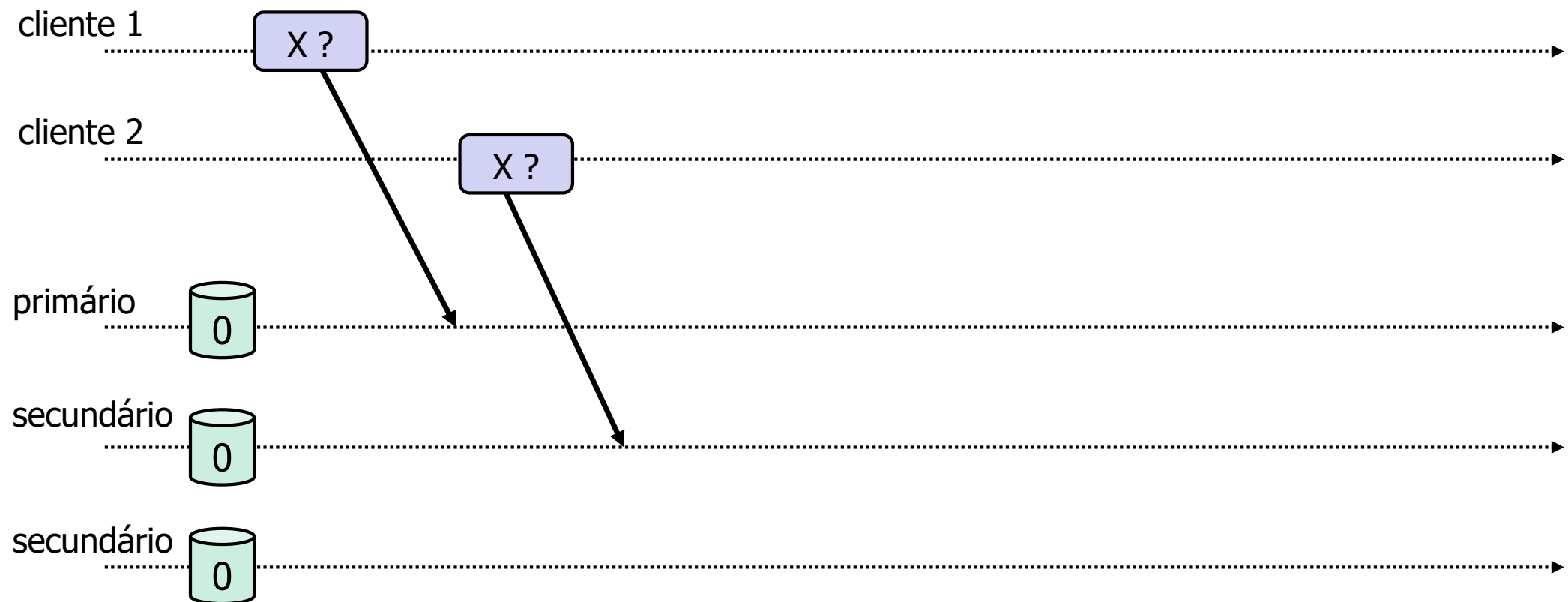
Objetivo: ter várias réplicas, mas que se comportem como se existisse apenas uma réplica que tolerasse falhas

Desafios:

1. Garantir que todas as réplicas convergem para o mesmo estado
- ~~2. Garantir que os clientes observam sempre a última escrita~~
2. **Garantir que os clientes observam sempre o resultados das suas escritas e que nunca observam um valor mais antigo do que o que observaram anteriormente.**

PRIMÁRIO / SECUNDÁRIO : LEITURA

Leituras são enviadas para qualquer uma das réplicas

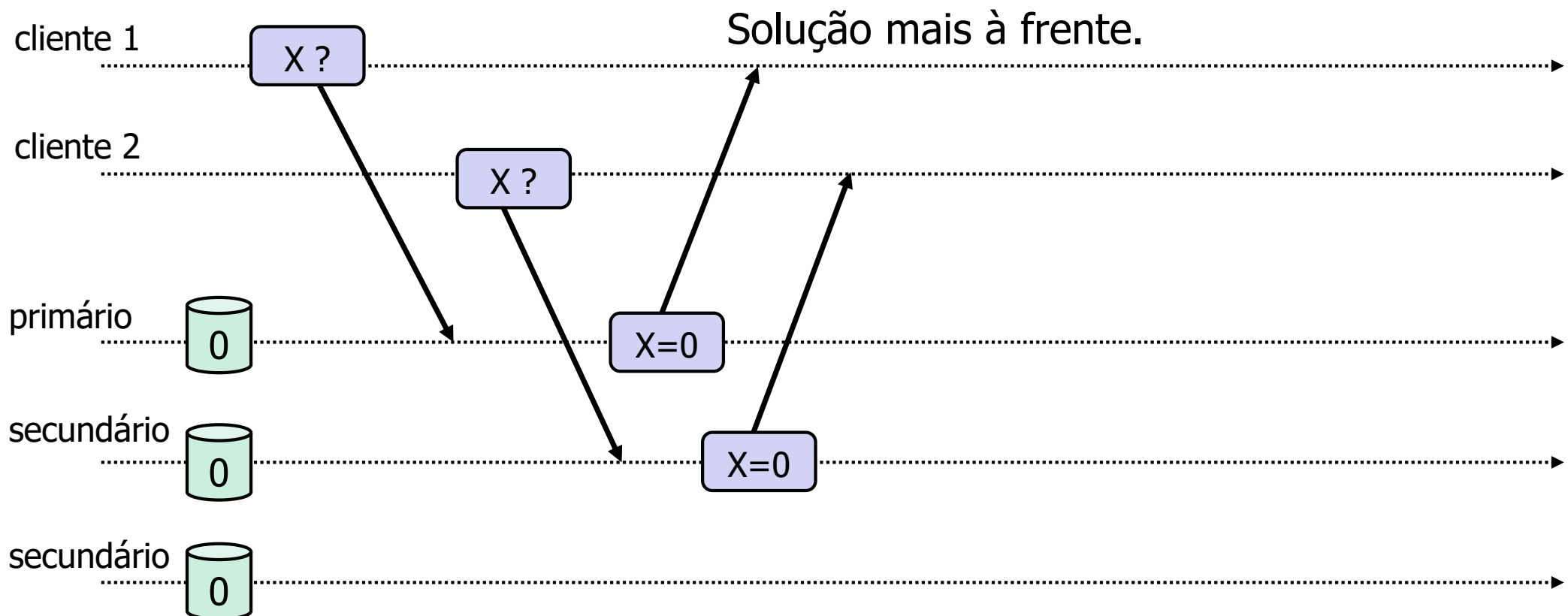


PRIMÁRIO / SECUNDÁRIO : LEITURA

Leituras são enviadas para qualquer uma das replicas

Secundário executa a operação de leitura e devolve o resultado

Com garantir que os clientes observam sempre o resultados das suas escritas e que nunca observam um valor mais antigo do que o que observaram anteriormente ?



PRIMÁRIO / SECUNDÁRIO : FALHAS

Modelo de falhas:

- as mensagens podem-se perder e trocar de ordem;
- um servidor pode falhar por crash, i.e., deixar de responder (e eventualmente recuperar).

PRIMÁRIO / SECUNDÁRIO : FALHAS

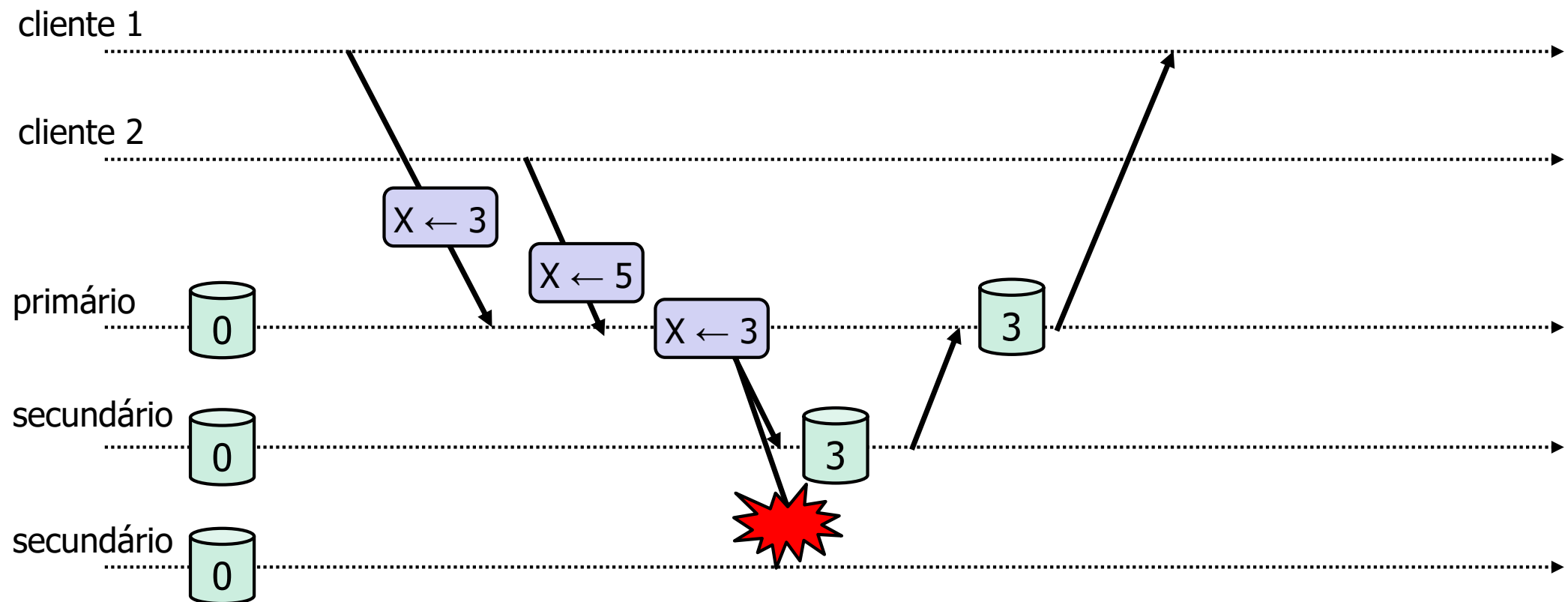
Modelo de falhas:

- **as mensagens podem-se perder e trocar de ordem;**
- um servidor pode falhar por crash, i.e., deixar de responder (e eventualmente recuperar).

PRIMÁRIO / SECUNDÁRIO : FALHA NAS MENSAGENS PARA UM SECUNDÁRIO

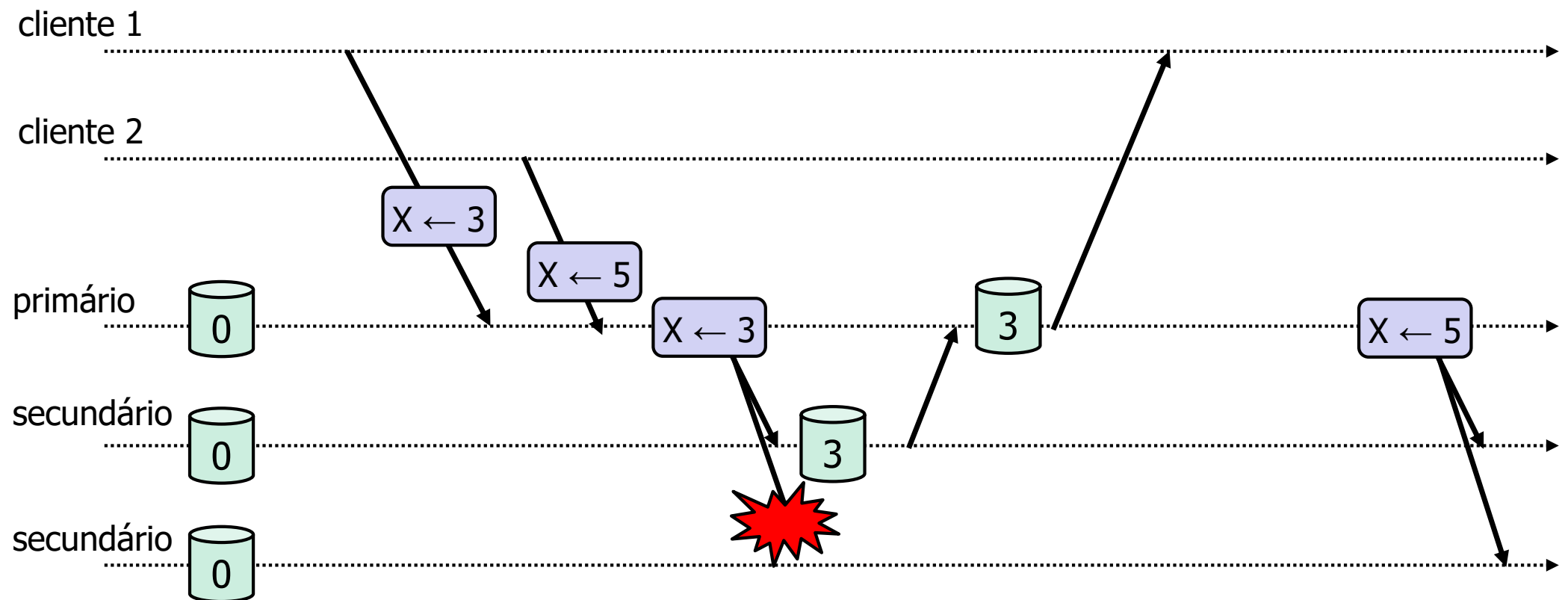
O que fazer quando uma mensagem para um secundário se perde?

O primário pode (e TEM de) devolver ACK ao cliente sem receber a resposta de todos os secundários. NOTA: isto é necessário porque o secundário pode ter falhado e não vai responder.



PRIMÁRIO / SECUNDÁRIO : FALHA NAS MENSAGENS
PARA UM SECUNDÁRIO

Como é que um secundário sabe que está a receber todas as operações?

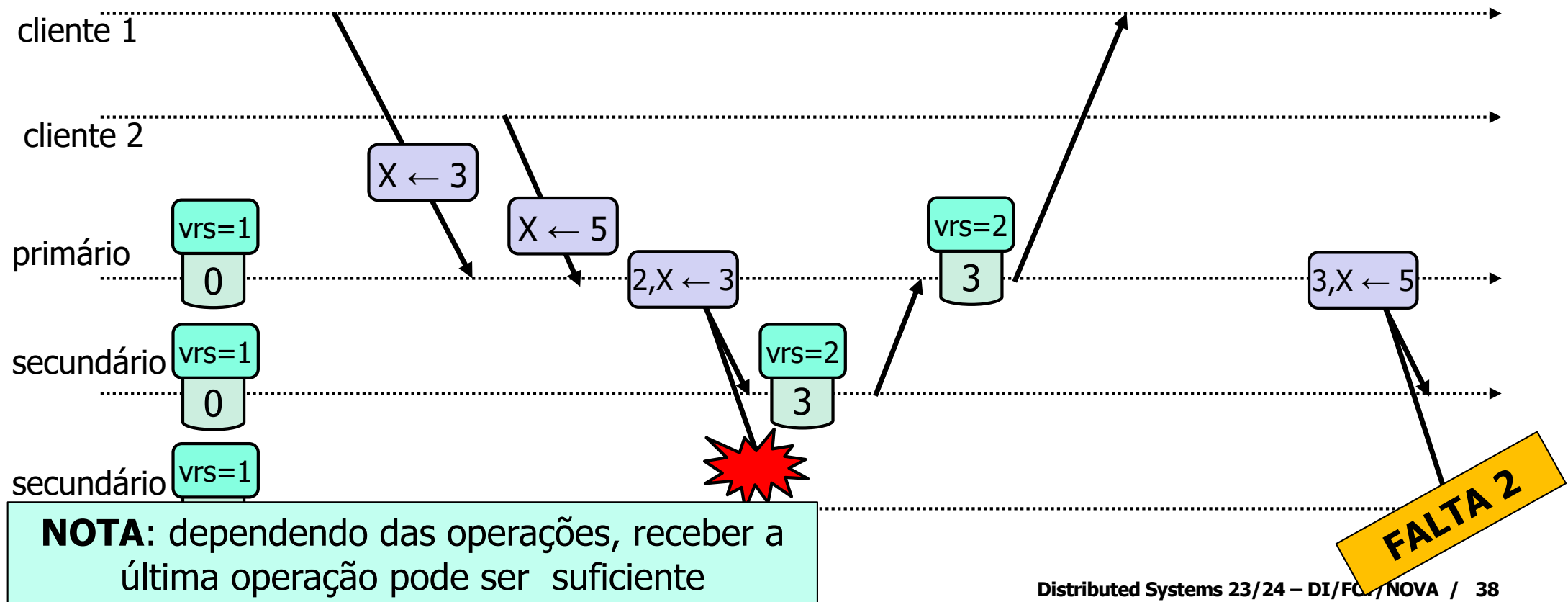


PRIMÁRIO / SECUNDÁRIO : FALHA NAS MENSAGENS PARA UM SECUNDÁRIO

Como é que um secundário sabe que está a receber todas as operações?

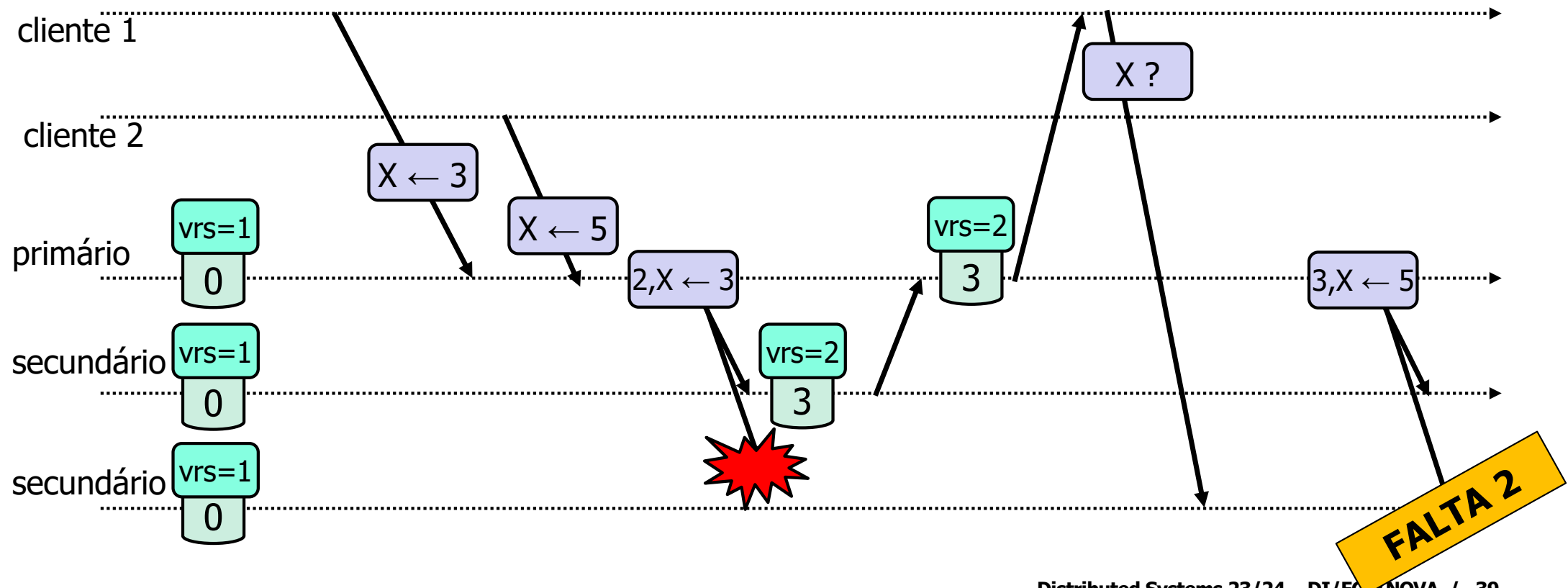
Operações devem ser numeradas pelo primário (canal FIFO) – este número pode ser usado para identificar a versão do estado.

Secundário deteta falta de operação na próxima operação (ou por contacto periódico com primário).



PRIMÁRIO / SECUNDÁRIO : FALHA NAS MENSAGENS PARA UM SECUNDÁRIO

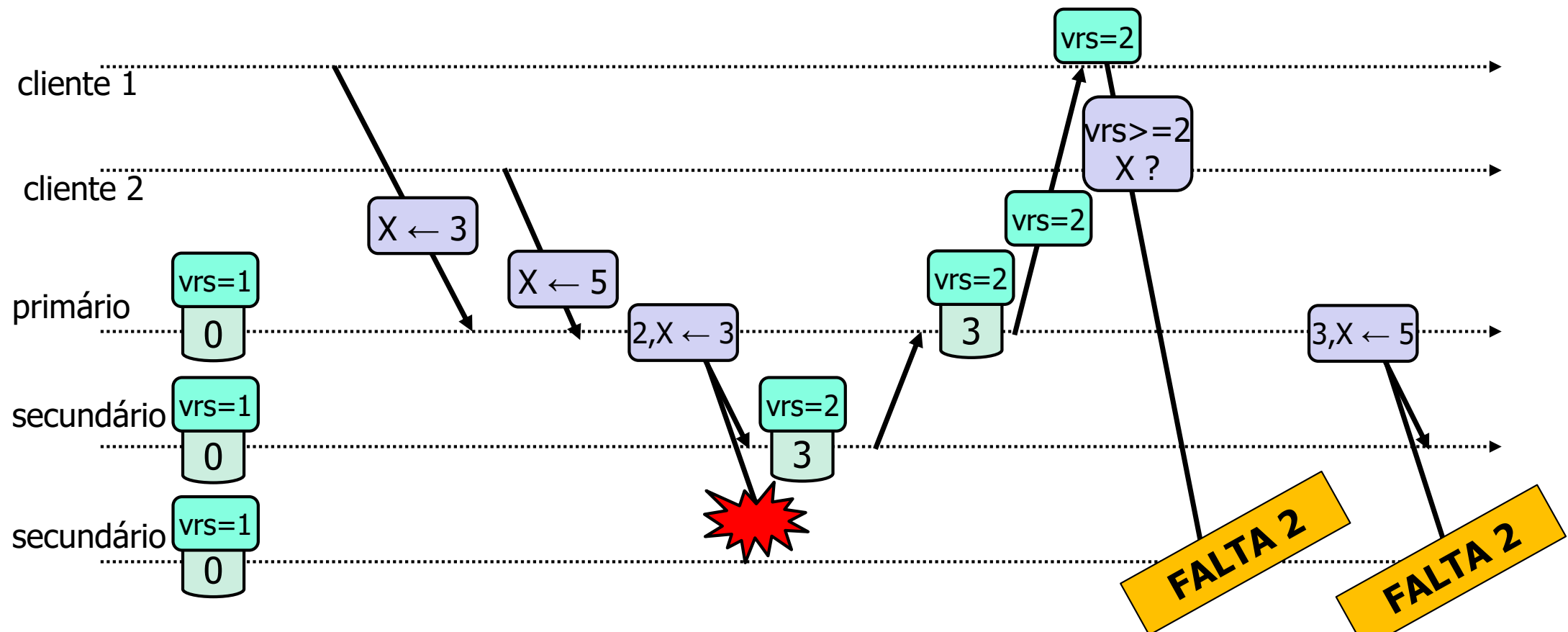
Se o clientes lerem do secundário, **como é que se garante que cliente não lê versão anterior à sua escrita?**



PRIMÁRIO / SECUNDÁRIO : FALHA NAS MENSAGENS PARA UM SECUNDÁRIO

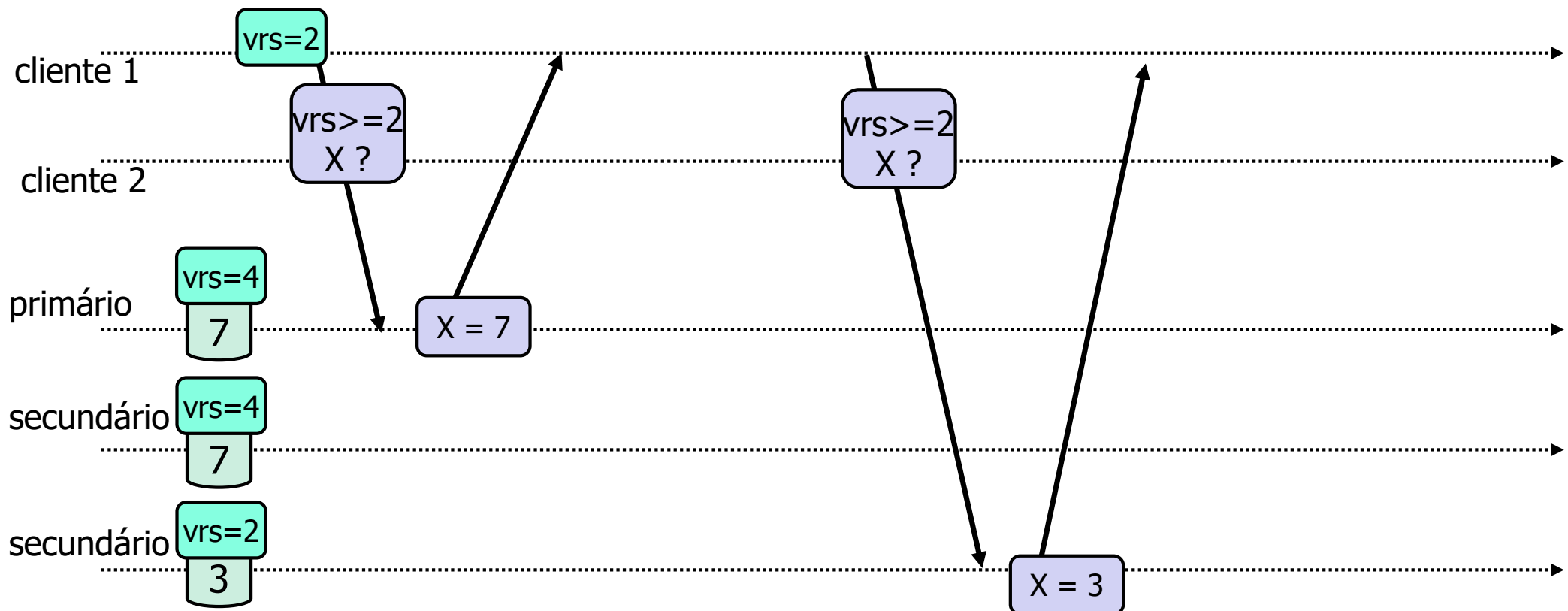
Se o clientes lerem do secundário, **como é que se garante que cliente não lê versão anterior à sua escrita?**

Cliente pode receber versão escrita e enviar essa versão no próximo pedido; apenas um servidor com uma versão igual ou superior responderá.



PRIMÁRIO / SECUNDÁRIO : FALHA NAS MENSAGENS PARA UM SECUNDÁRIO

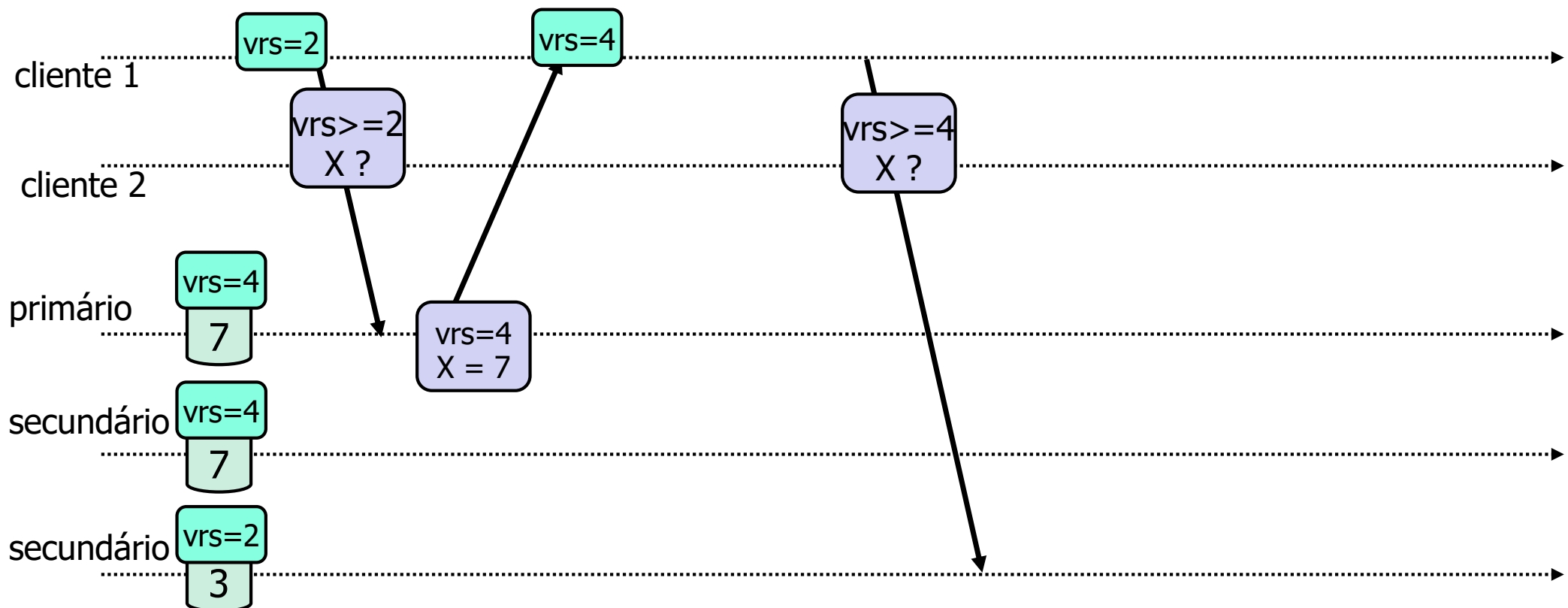
Se o clientes lerem do secundário, **como é que se garante que cliente não lê versão anterior à da última leitura?**



PRIMÁRIO / SECUNDÁRIO : FALHA NAS MENSAGENS PARA UM SECUNDÁRIO

Se o clientes lerem do secundário, **como é que se garante que cliente não lê versão anterior à da última leitura?**

Cliente pode receber versão lida e enviar essa versão no próximo pedido; apenas um servidor com uma versão igual ou superior responderá.



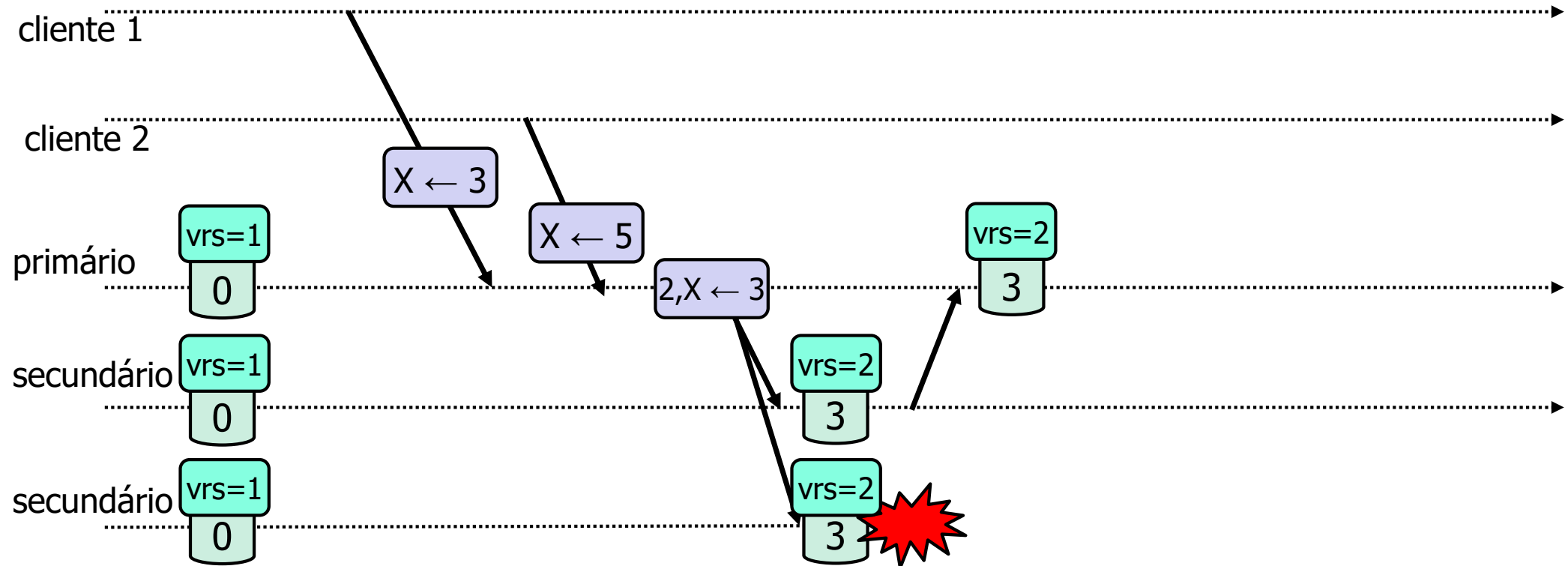
PRIMÁRIO / SECUNDÁRIO : FALHAS

Modelo de falhas:

- as mensagens podem-se perder e trocar de ordem;
- **um servidor pode falhar por crash, i.e., deixar de responder (e eventualmente recuperar).**

PRIMÁRIO / SECUNDÁRIO : FALHA NUM SECUNDÁRIO

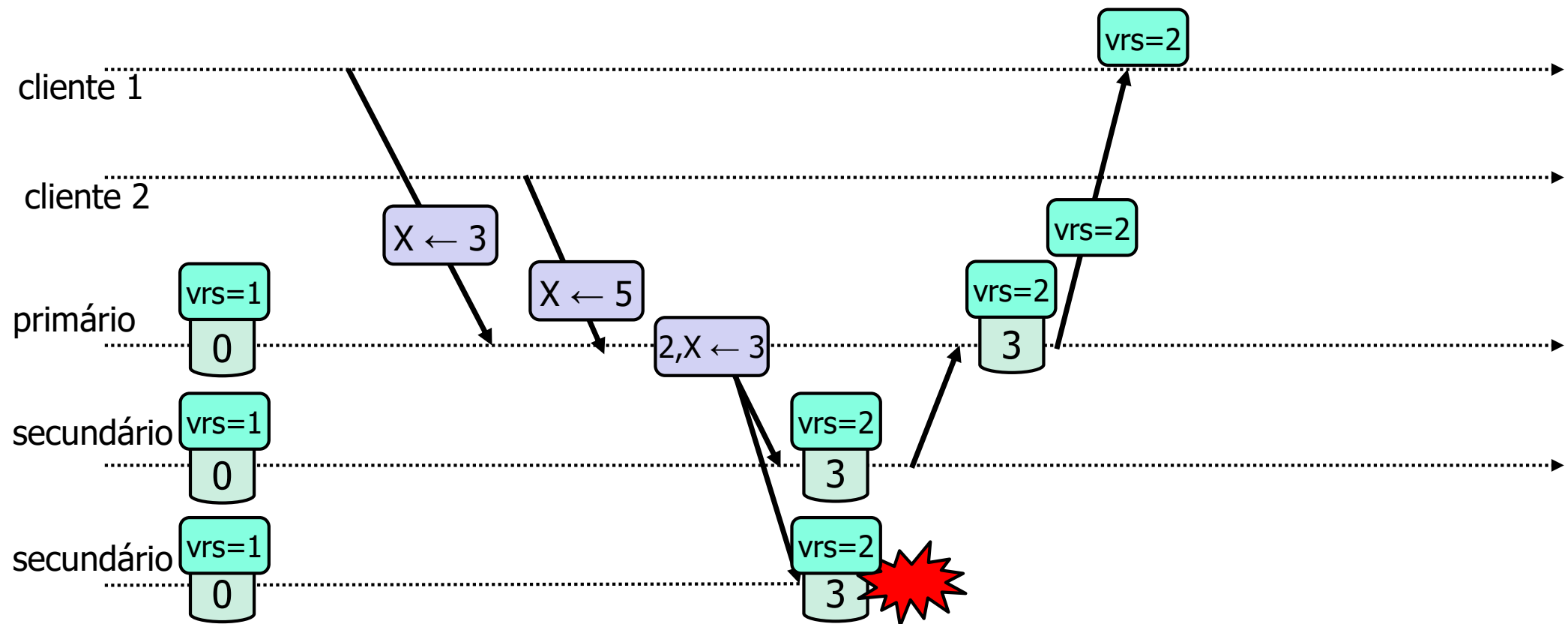
Como se lida com a falha num secundário?



PRIMÁRIO / SECUNDÁRIO : FALHA NUM SECUNDÁRIO

Como se lida com a falha num secundário?

Da mesma forma que com as falhas de comunicação com o secundário. O primário e os outros secundários continuam a executar.

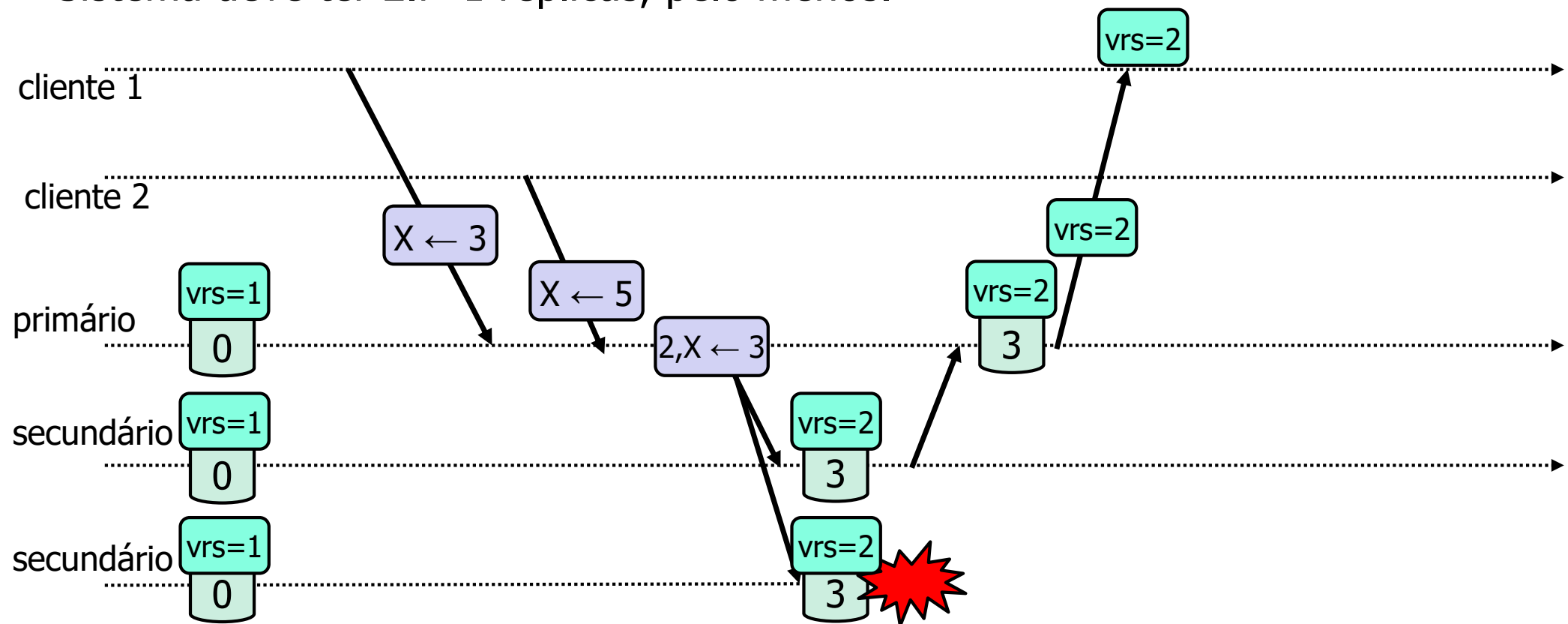


PRIMÁRIO / SECUNDÁRIO : FALHA NUM SECUNDÁRIO

Quantos secundários precisam de responder de forma a que uma escrita não se perca?

f secundários para tolerar f falhas.

Sistema deve ter $2.f+1$ réplicas, pelo menos.

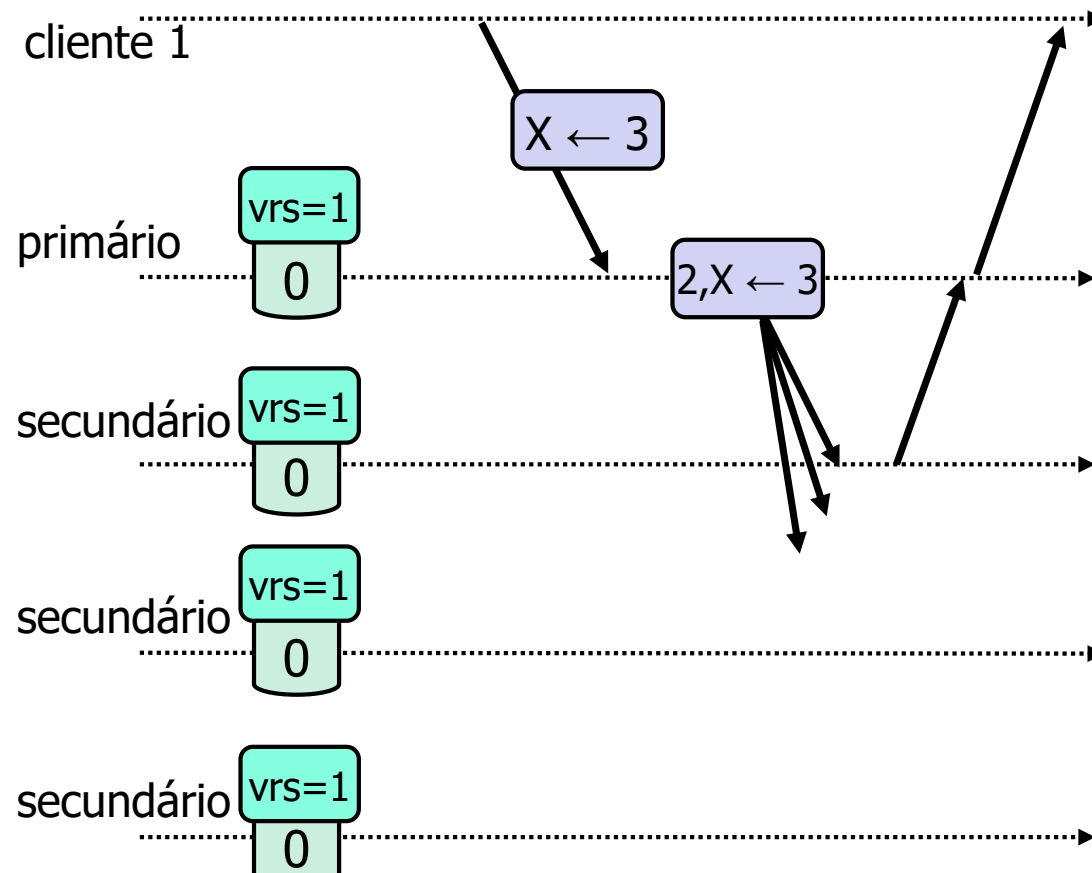


PRIMÁRIO / SECUNDÁRIO : FALHA NUM SECUNDÁRIO

Porquê $2.f+1$ réplicas para tolerar f falhas?

Vamos supor que tínhamos apenas 4 réplicas para tolerar 2 falhas. Neste caso, o primário tem de responder ao cliente assim que recebe 1 ack dum secundário (porque os outros secundários podem ter falhado e nunca iriam responder).

Problema?



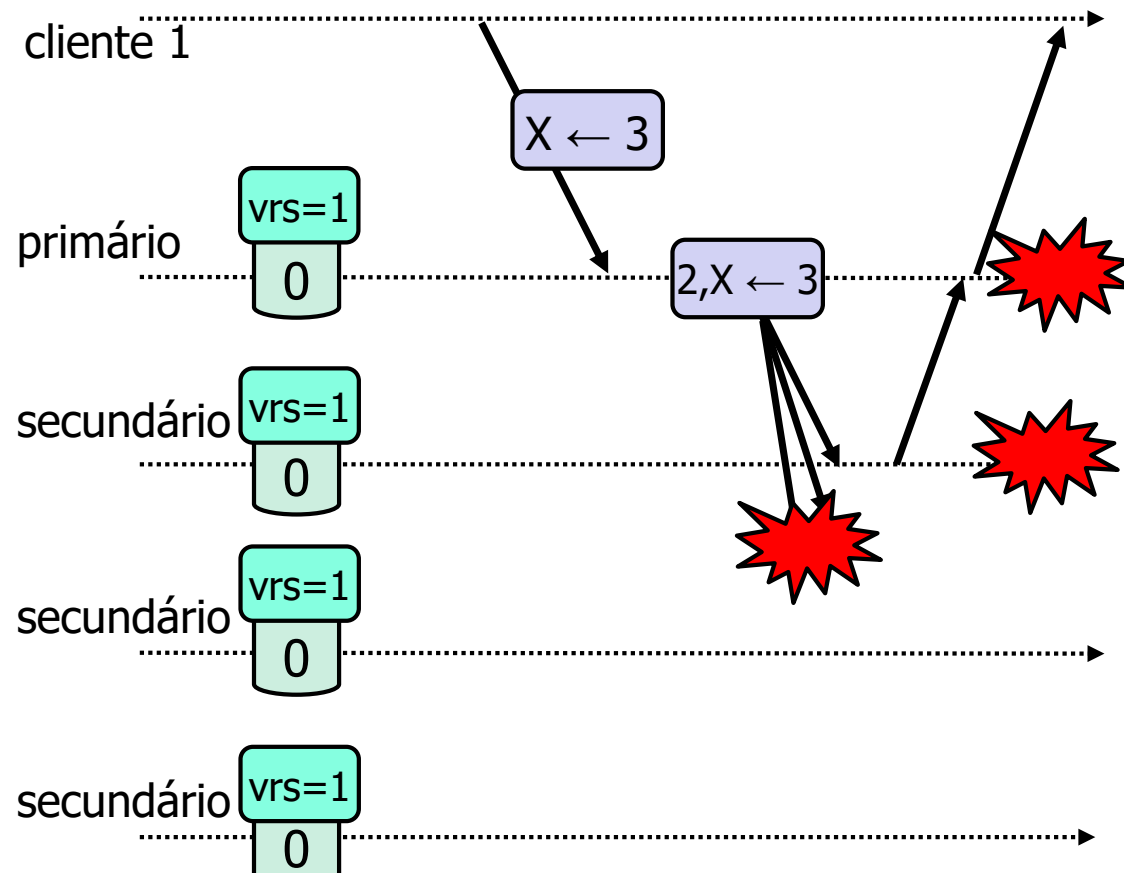
PRIMÁRIO / SECUNDÁRIO : FALHA NUM SECUNDÁRIO

Porquê $2.f+1$ réplicas para tolerar f falhas?

Vamos supor que tínhamos apenas 4 réplicas para tolerar 2 falhas. Neste caso, o primário tem de responder ao cliente assim que recebe 1 ack dum secundário (porque os outros secundários podem ter falhado e nunca iriam responder).

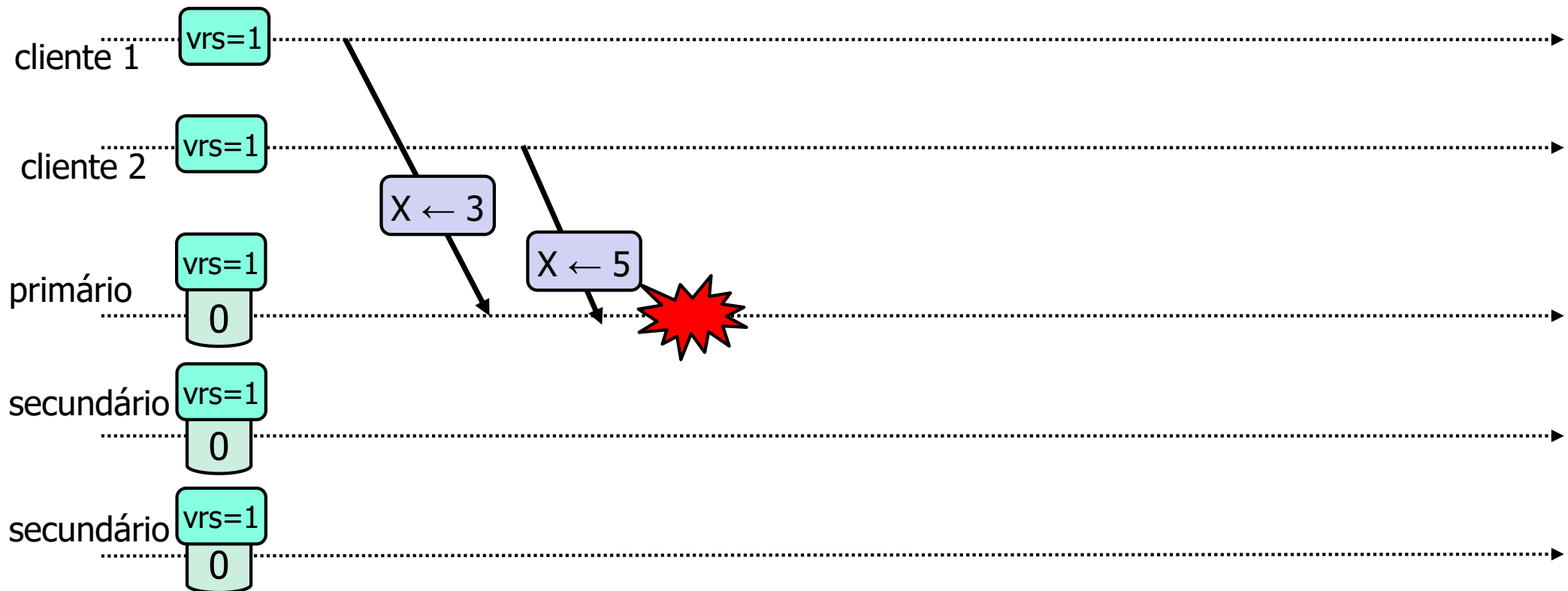
Problema?

Se primário e secundário que respondeu falharem a seguir, a operação perde-se.



PRIMÁRIO / SECUNDÁRIO : FALHA NO PRIMÁRIO

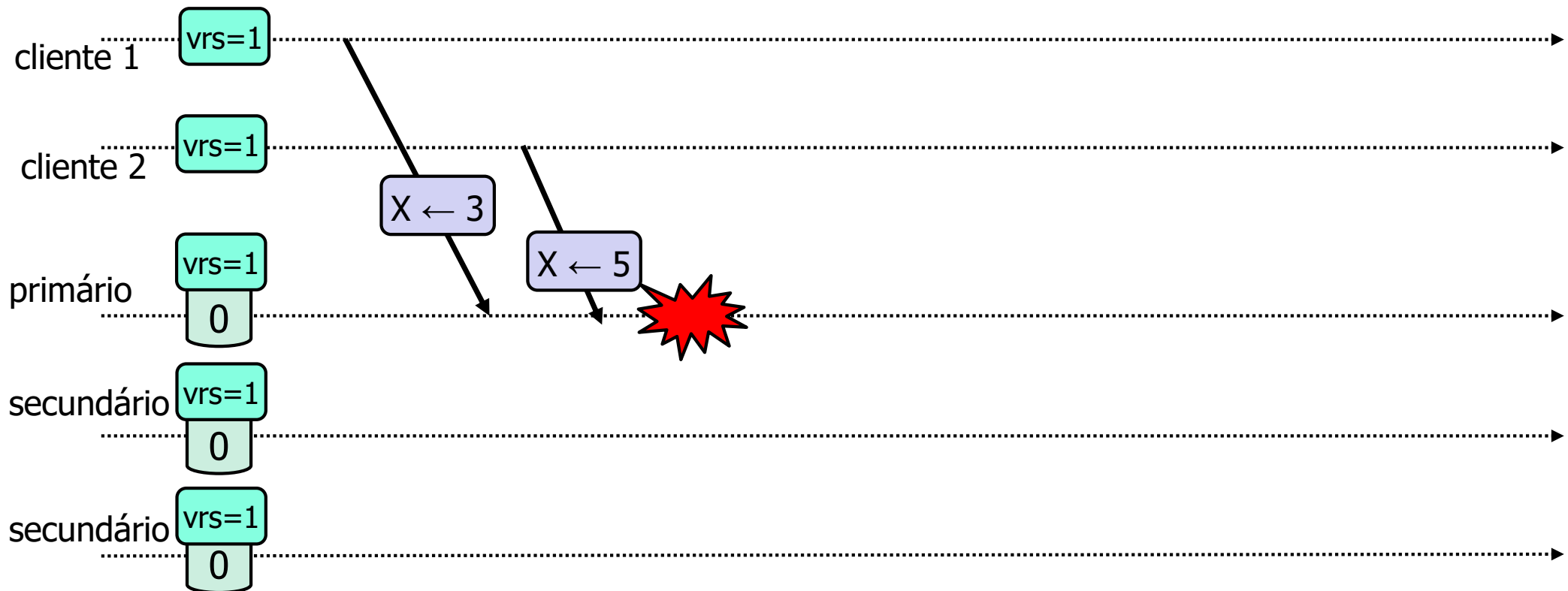
O que fazer quando o primário falha?



PRIMÁRIO / SECUNDÁRIO : FALHA NO PRIMÁRIO

O que fazer quando o primário falha?

HIP. 1: Parar de aceitar escritas. Neste caso não se está a tolerar 1 falha.

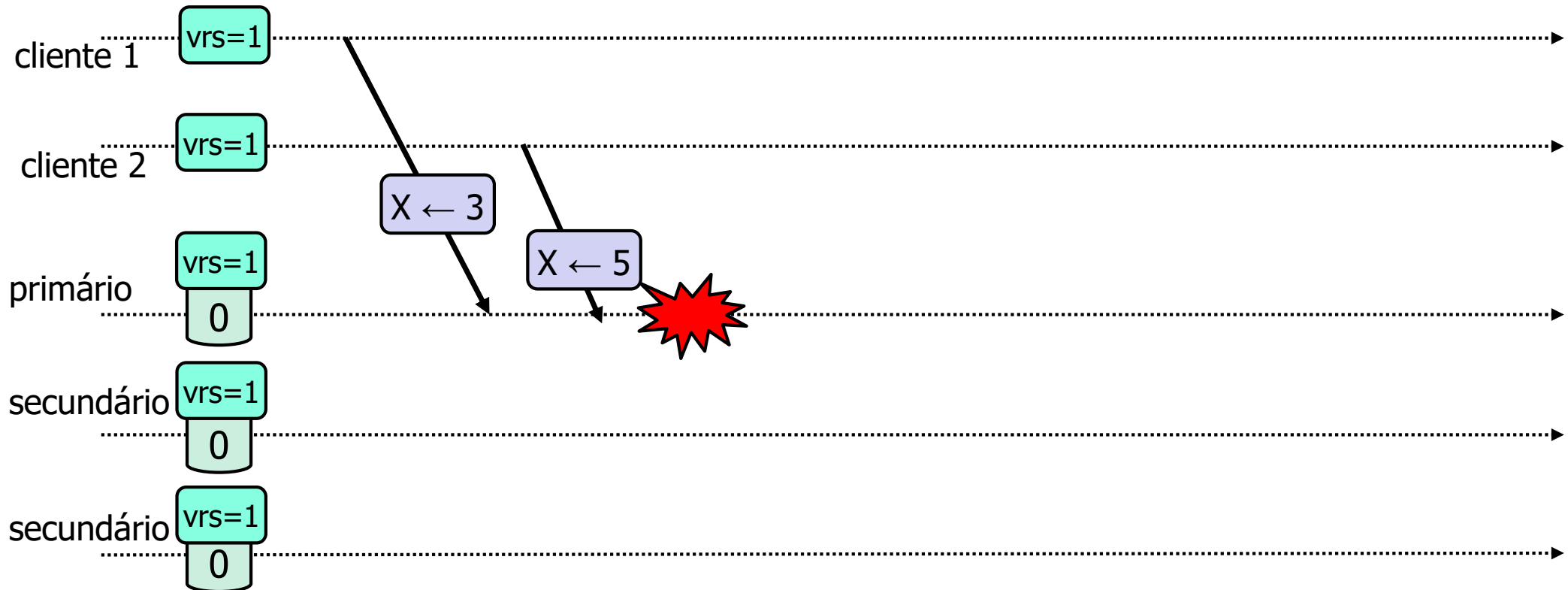


PRIMÁRIO / SECUNDÁRIO : FALHA NO PRIMÁRIO

O que fazer quando o primário falha?

Hip. 2: Continuar a executar.

1. Um secundário deve ser eleito o novo primário
2. Secundários devem concordar em qual a última operação executada (e possivelmente confirmada a um cliente)

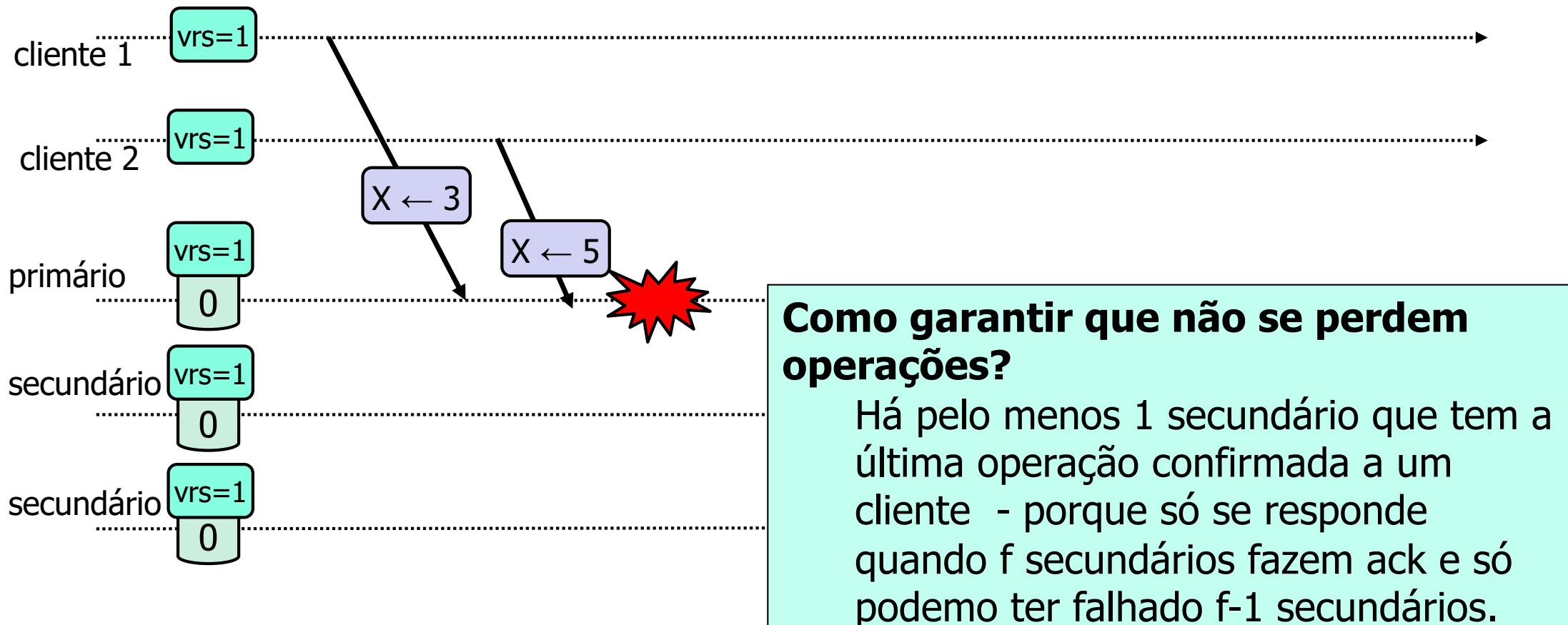


PRIMÁRIO / SECUNDÁRIO : FALHA NO PRIMÁRIO

O que fazer quando o primário falha?

Hip. 2: Continuar a executar.

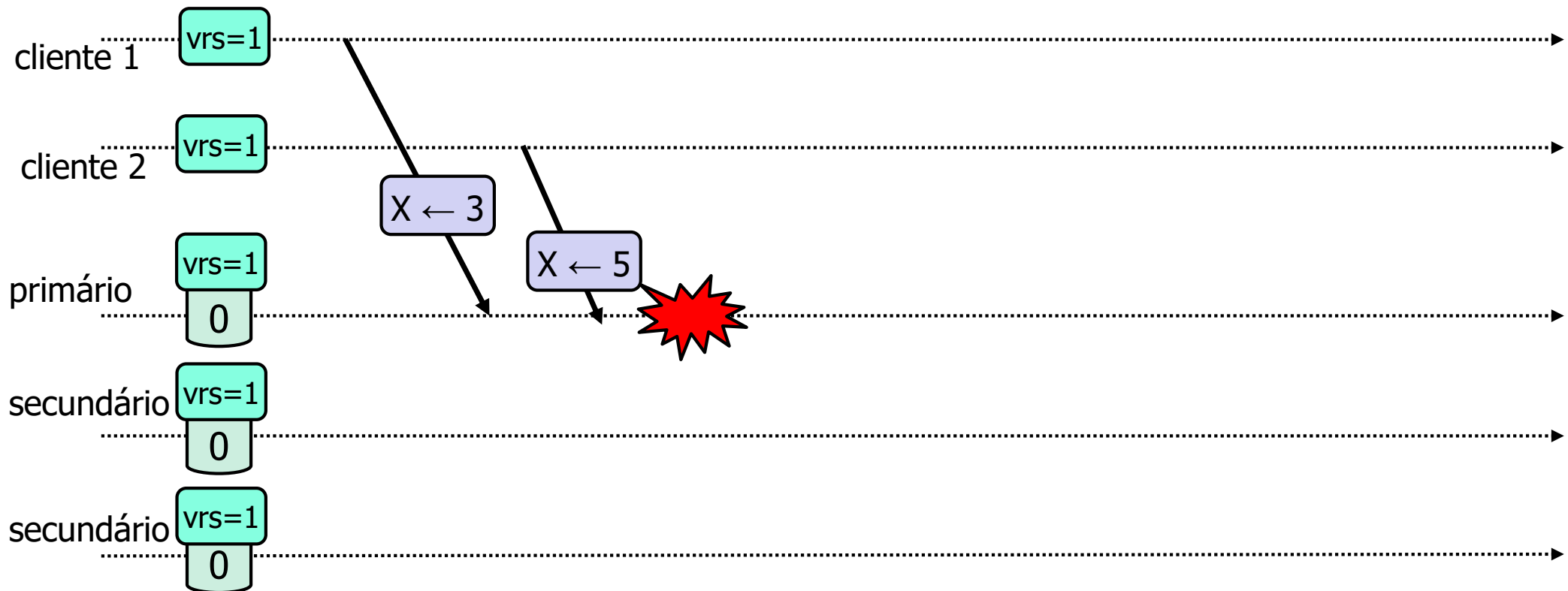
1. Um secundário deve ser eleito o novo primário
2. **Secundários devem concordar em qual a última operação executada (e possivelmente confirmada a um cliente)**



PRIMÁRIO / SECUNDÁRIO : FALHA NO PRIMÁRIO

Como eleger um novo primário?

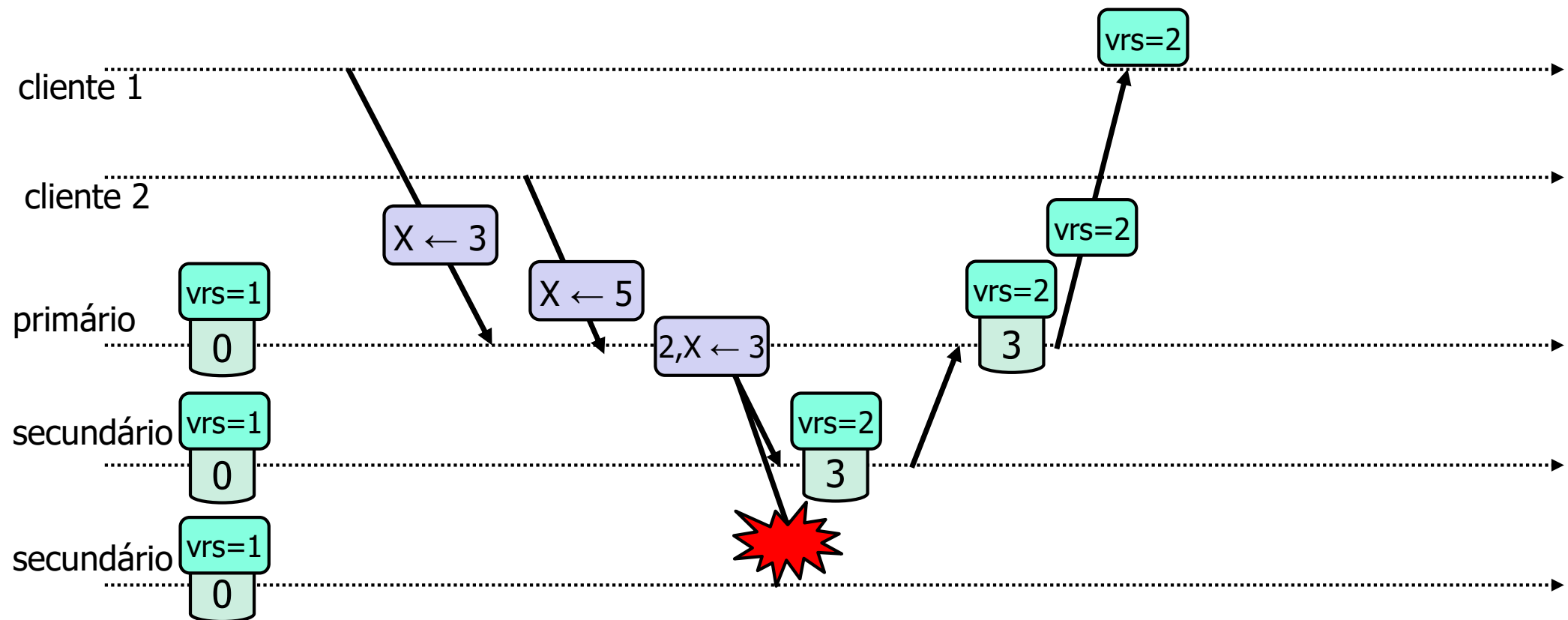
- Algoritmo de eleição, sistema de comunicação em grupo, etc.
- [algoritmos estudados em ASD]



NA ÚLTIMA AULA: PRIMÁRIO / SECUNDÁRIO

Operações de escrita enviadas para o primário, que ordena as operações.

Primário propaga operação para secundário e espera por f confirmações antes de executar a operação.

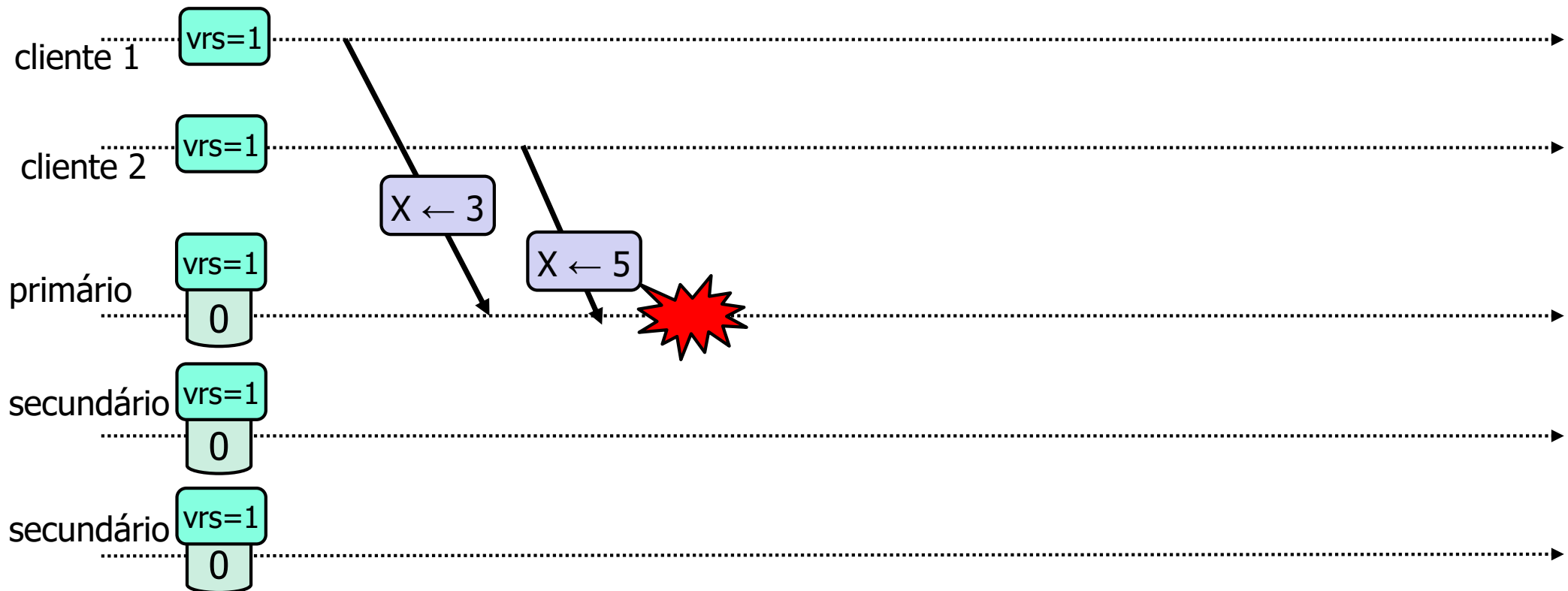


PRIMÁRIO / SECUNDÁRIO : FALHA NO PRIMÁRIO

O que fazer quando o primário falha?

HIP. 1: Parar de aceitar escritas. Neste caso não se está a tolerar 1 falha.

HIP. 2: Continuar a executar. Necessário eleger um novo primário.



APACHE ZOOKEEPER

O Zookeeper é um sistema de coordenação centralizado e replicado com elevada disponibilidade.

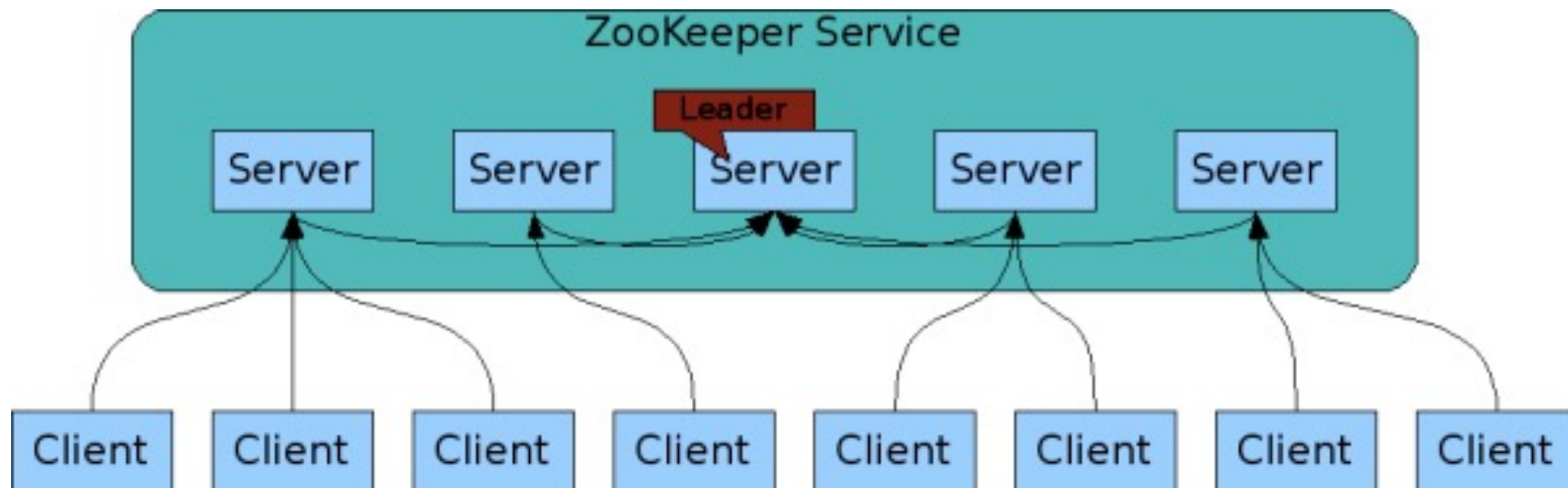
Usado para manter informação de configuração.

"ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services."

ZOOKEEPER: ARQUITETURA

Conjunto de servidores mantém réplicas da “base de dados”.

Operações totalmente ordenadas.



ZOOKEEPER: MODELO DE DADOS

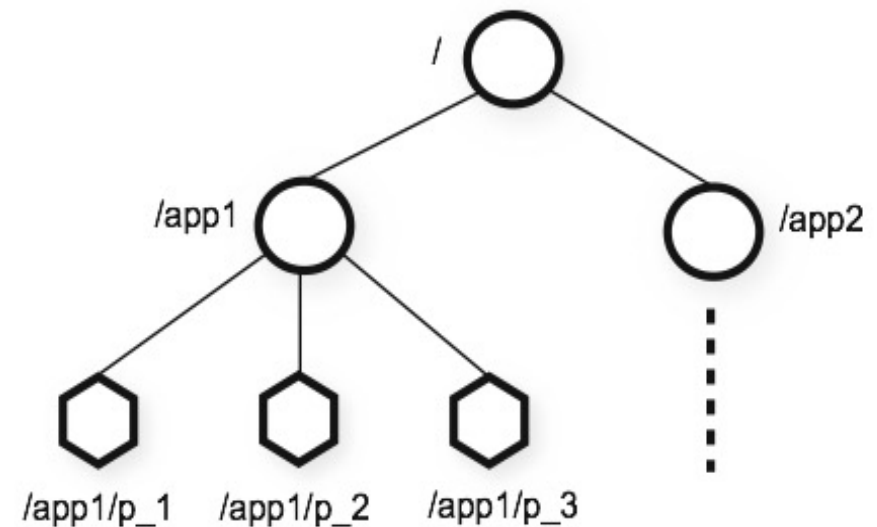
Nós organizados numa estrutura hierárquica

“ZooKeeper was designed to store coordination data: status information, configuration, location information, etc., so the data stored at each node is usually small, in the byte to kilobyte range.”

Tipos de nós

Permanentes

Efémeros – que apenas existem enquanto o nó que os criou está a executar



ZOOKEEPER: OPERAÇÕES

Criar nó

Permanente ou efêmero

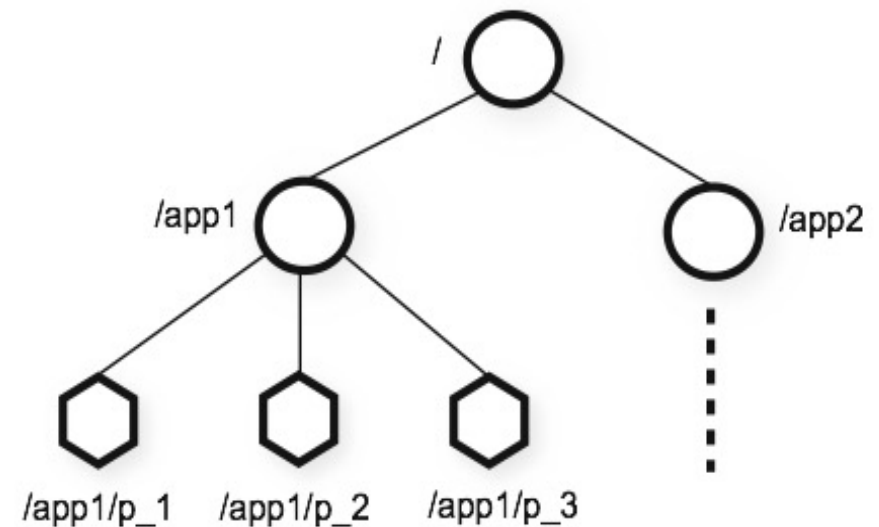
Normal ou sequencial (um nó sequencial é um nó ao qual é adicionado um número de sequência ao nome)

Ler / escrever valor do nó

Escrever condicionalmente

Remover nó

Notificação de alteração – pode-se pedir para ser notificado de alterações no sistema



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

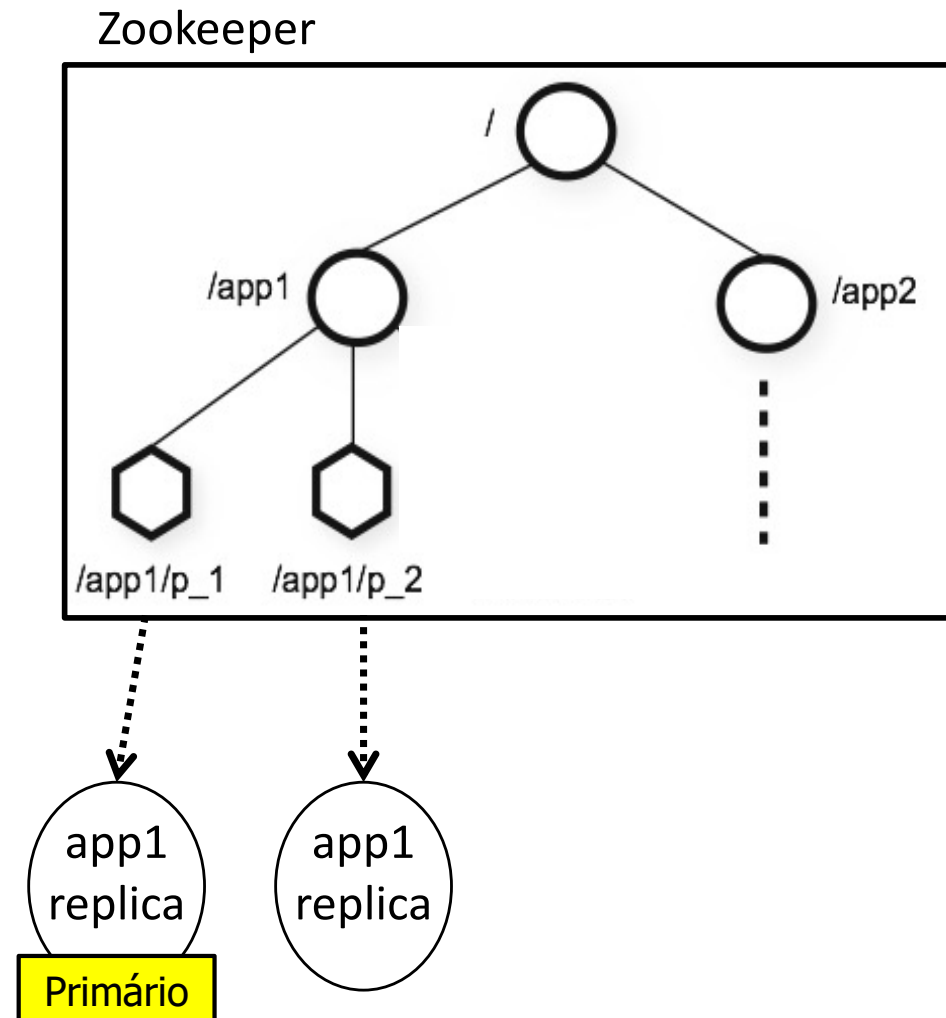
Cria-se diretoria para o serviço (e.g. `/app1`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

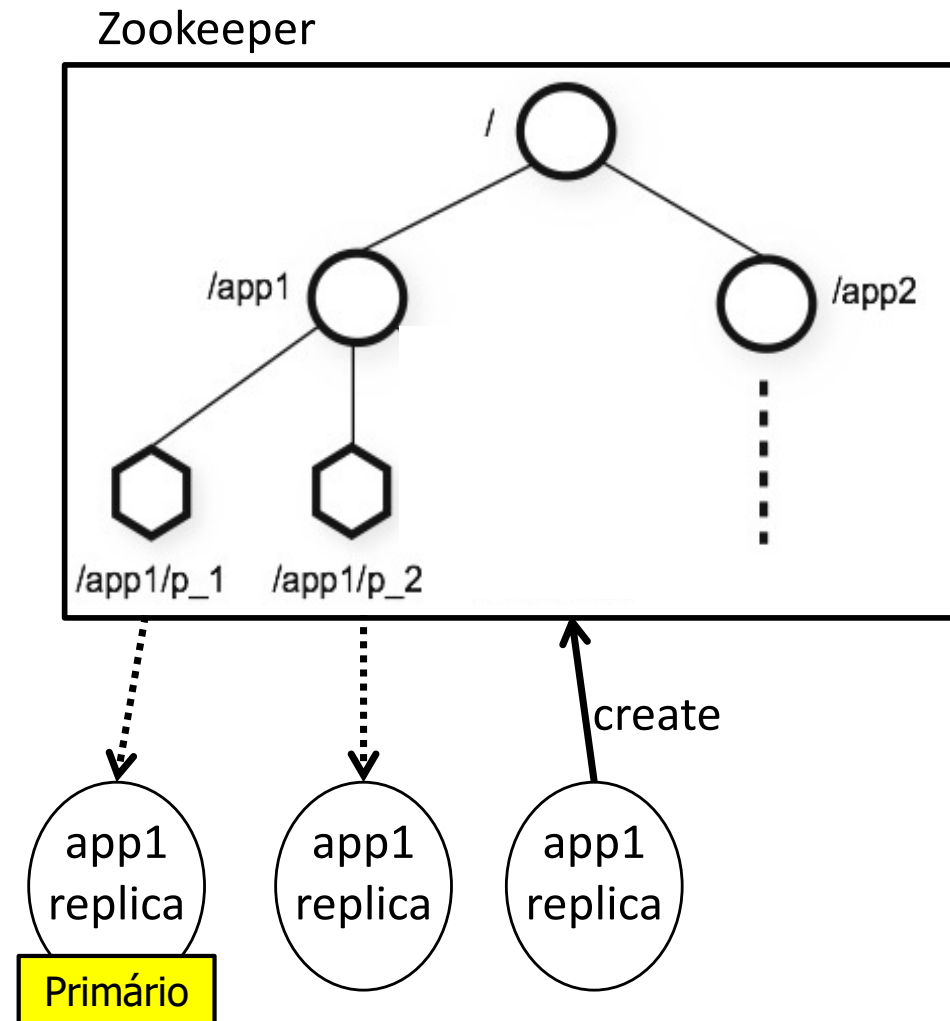
Cria-se diretoria para o serviço (e.g. `"/app1"`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `"/app1/p_"`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

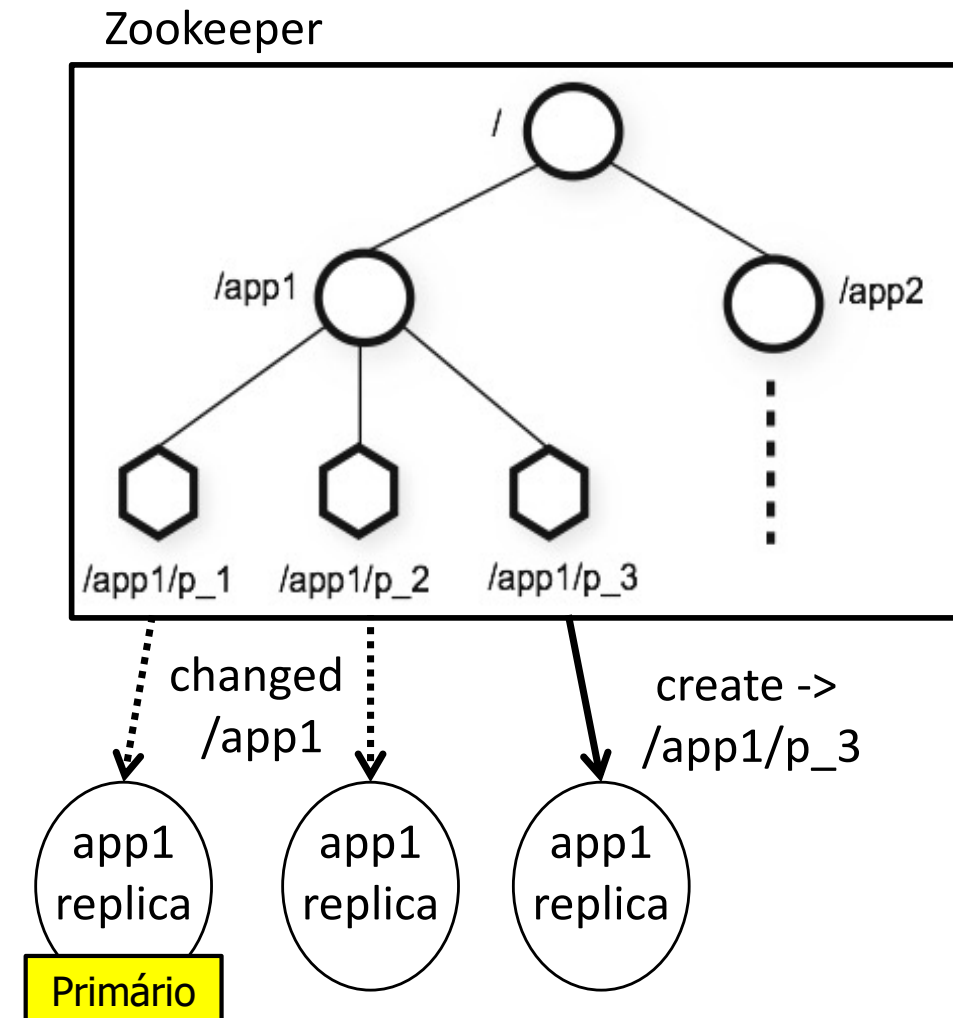
Cria-se diretoria para o serviço (e.g. `/app1`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

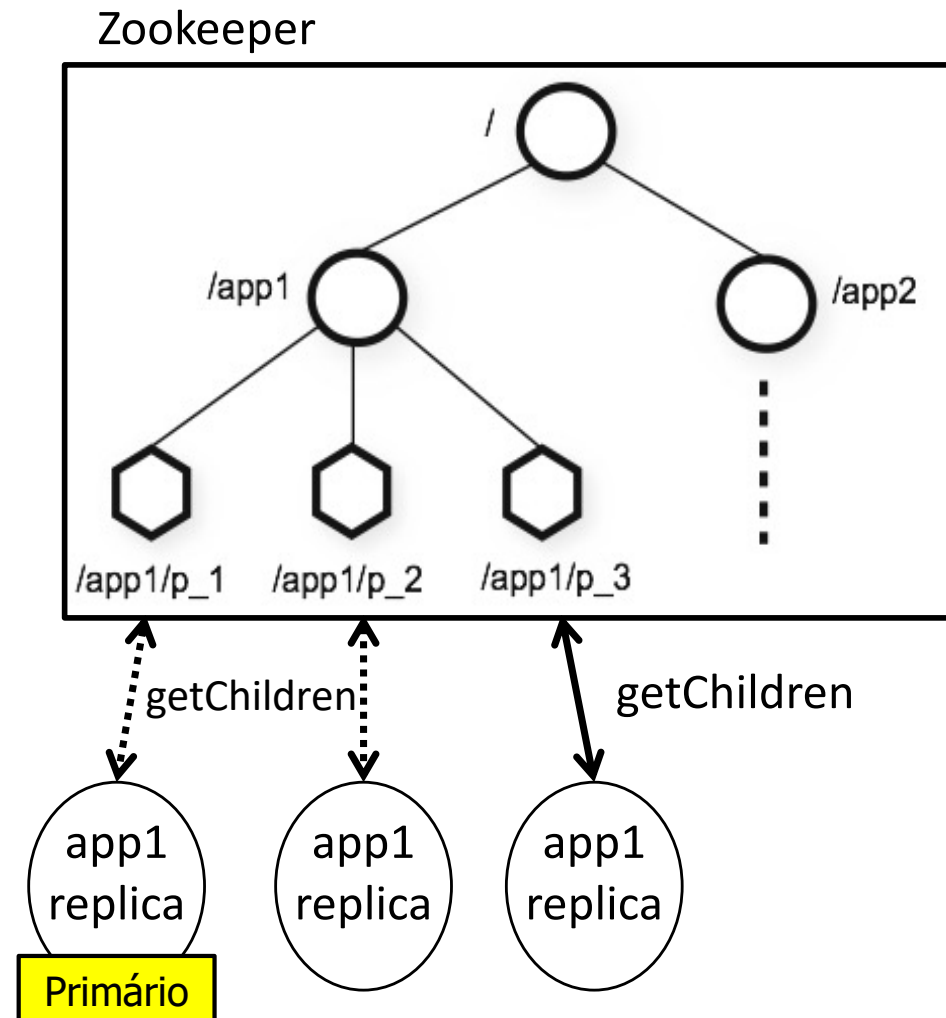
Cria-se diretoria para o serviço (e.g. `/app1`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

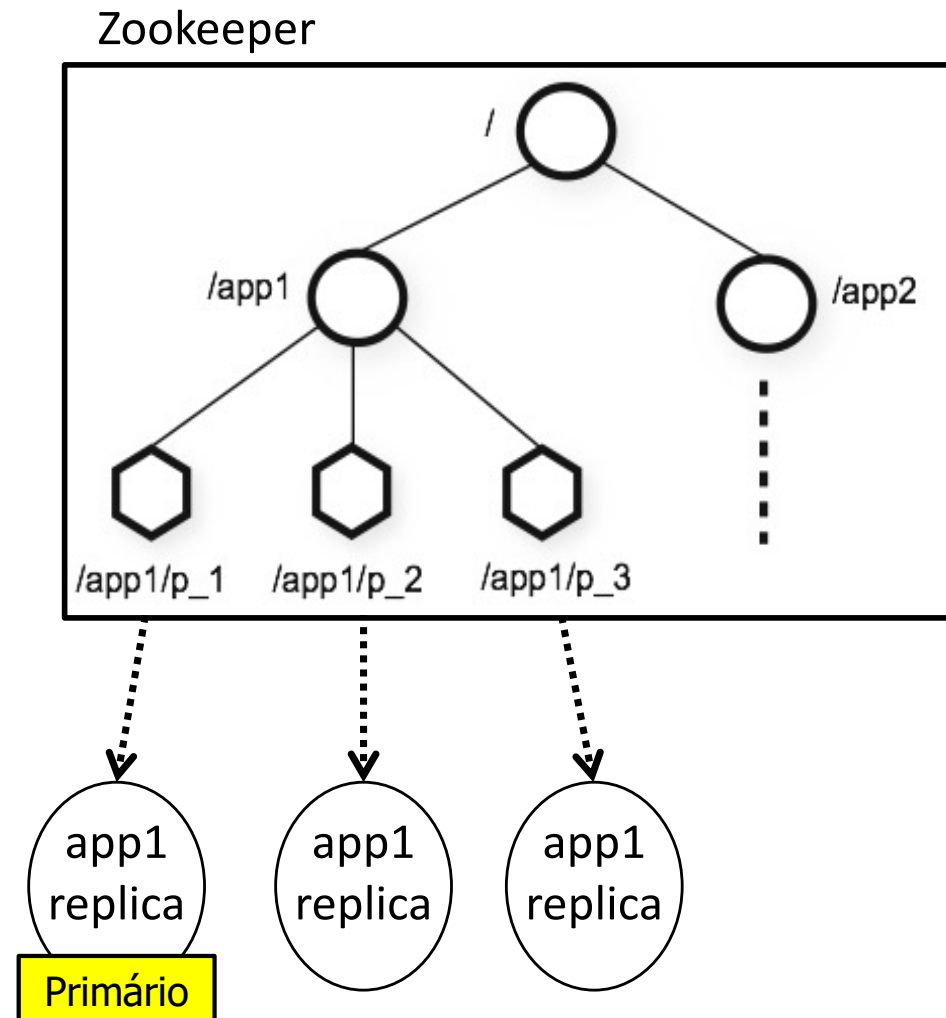
Cria-se diretoria para o serviço (e.g. `/app1`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

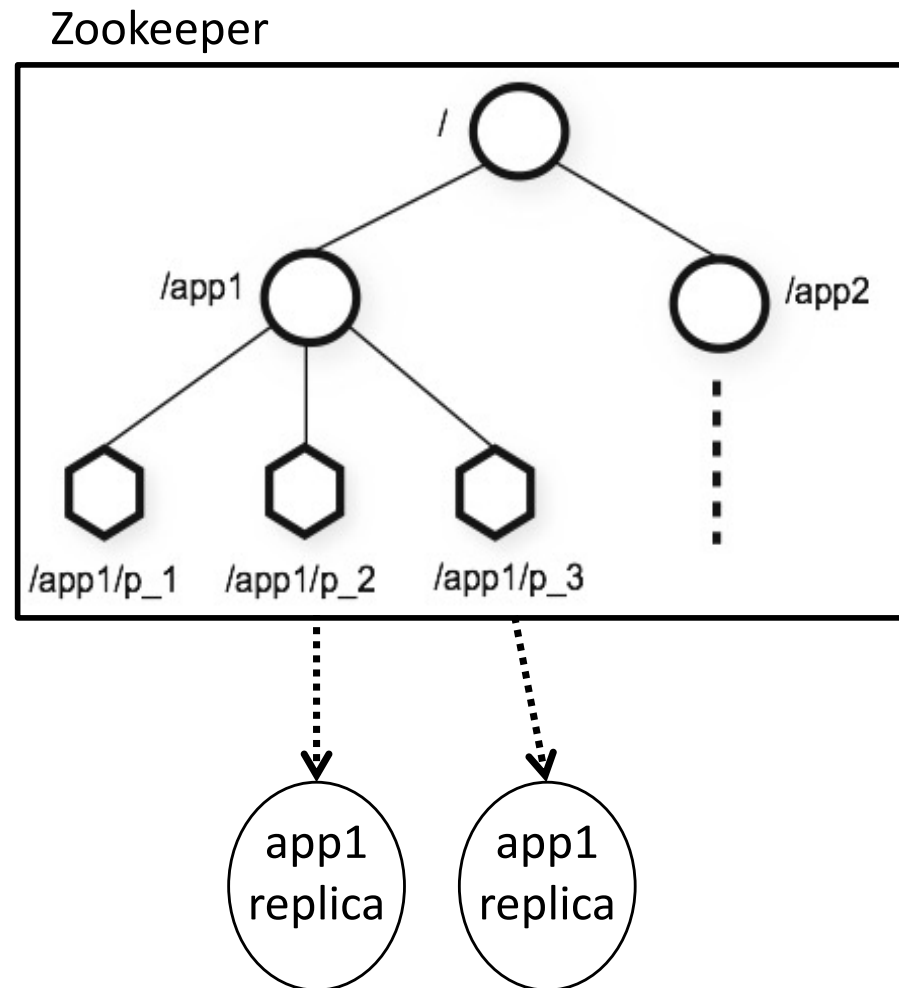
Cria-se diretoria para o serviço (e.g. `/app1`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

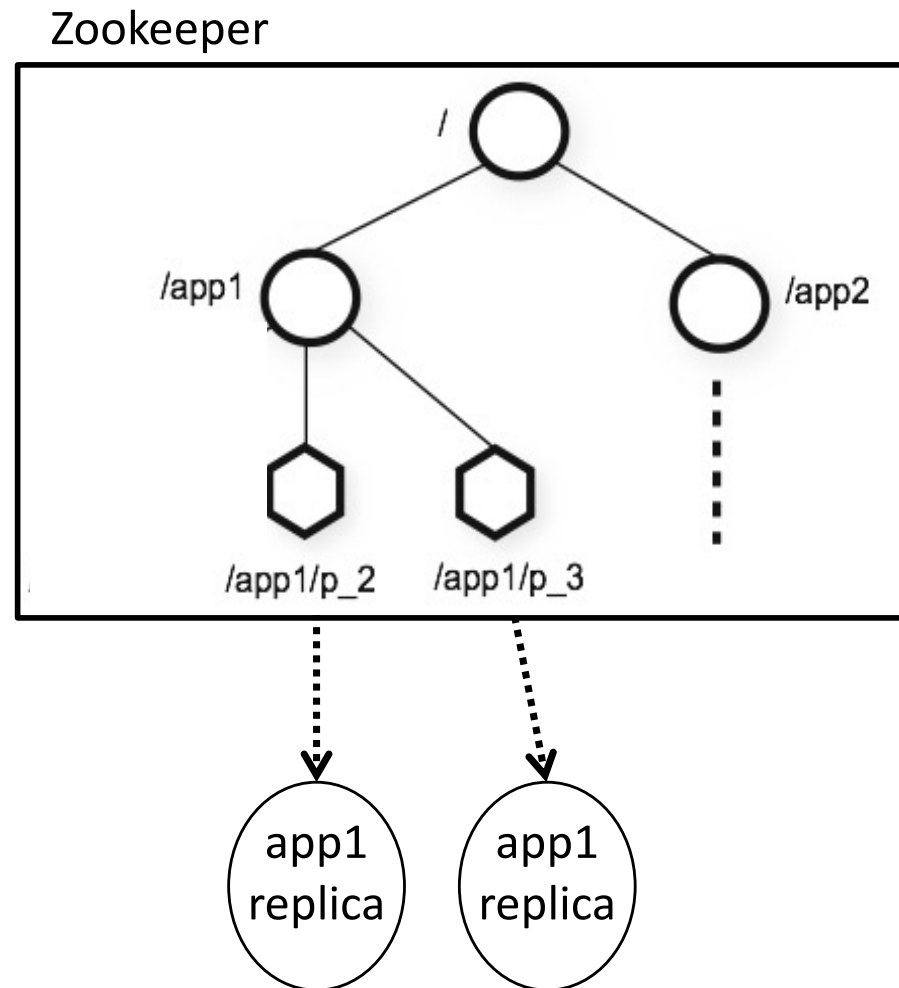
Cria-se diretoria para o serviço (e.g. `/app1`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

Cria-se diretoria para o serviço (e.g. `/app1`)

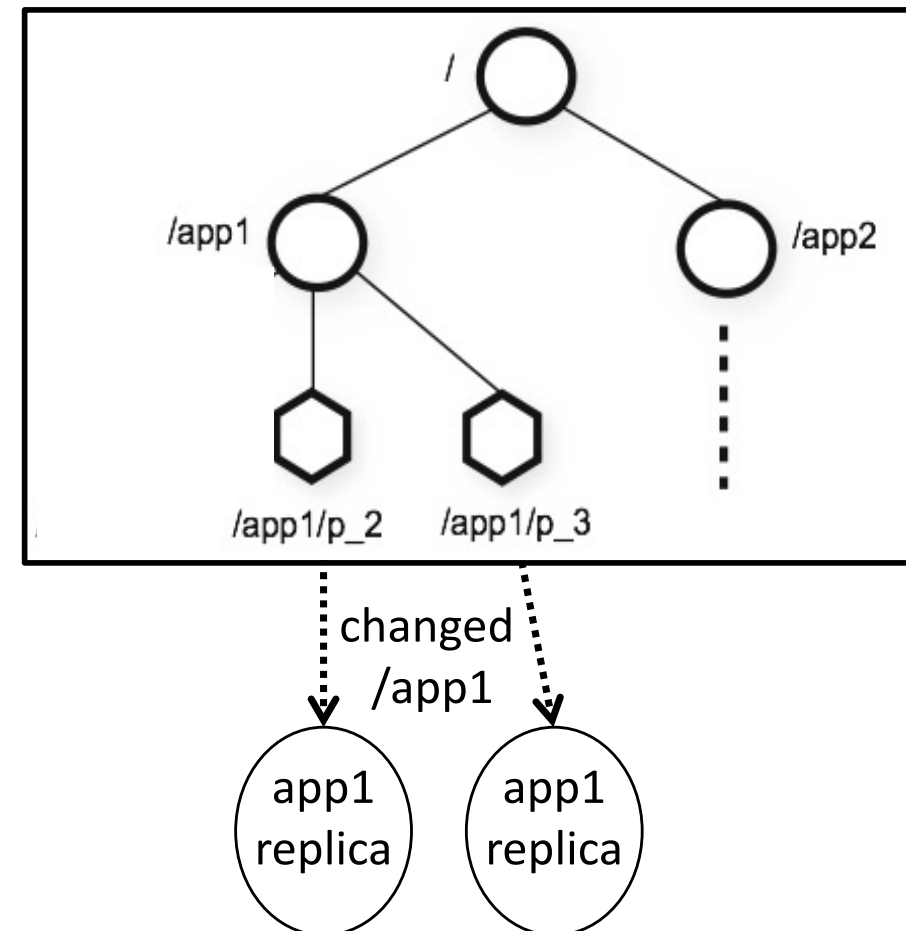
Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Zookeeper



ELEIÇÃO DO PRIMÁRIO: ALGORITMO

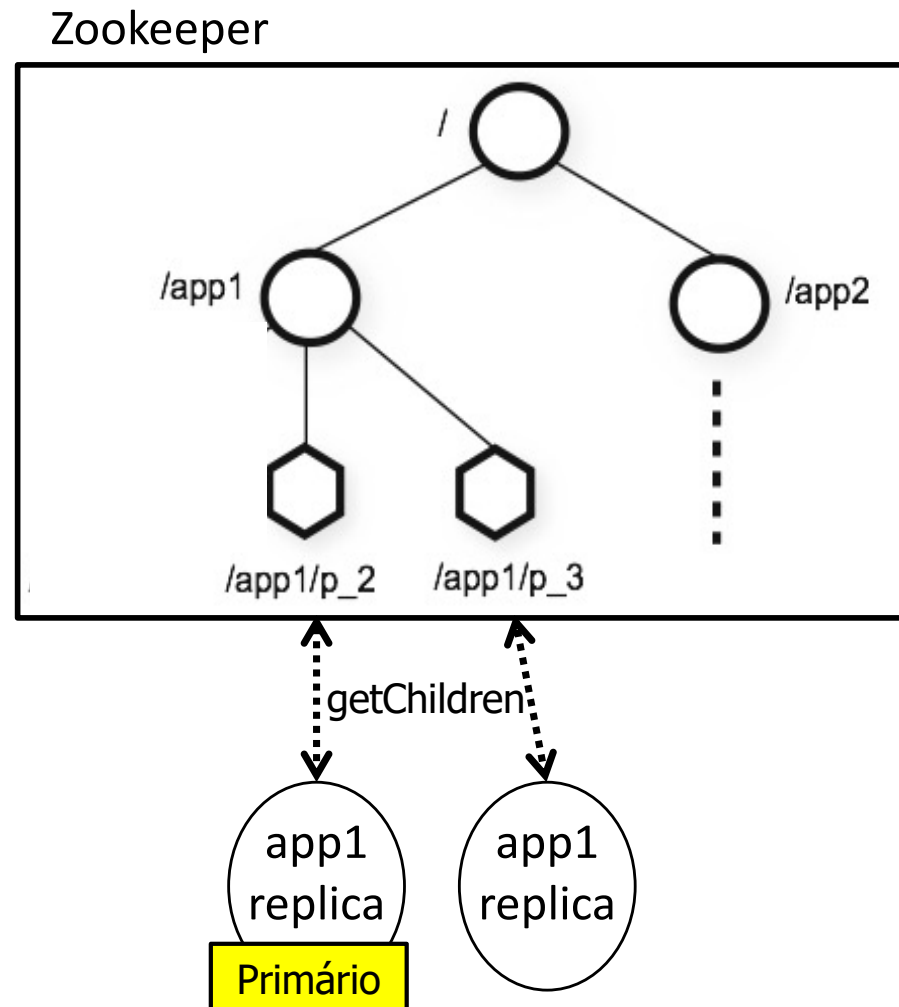
Cria-se diretoria para o serviço (e.g. `/app1`)

Um servidor:

1. cria um nó na diretoria, com o tipo efêmero e sequencial (e.g. `/app1/p_`)
2. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário

Quando há uma alteração:

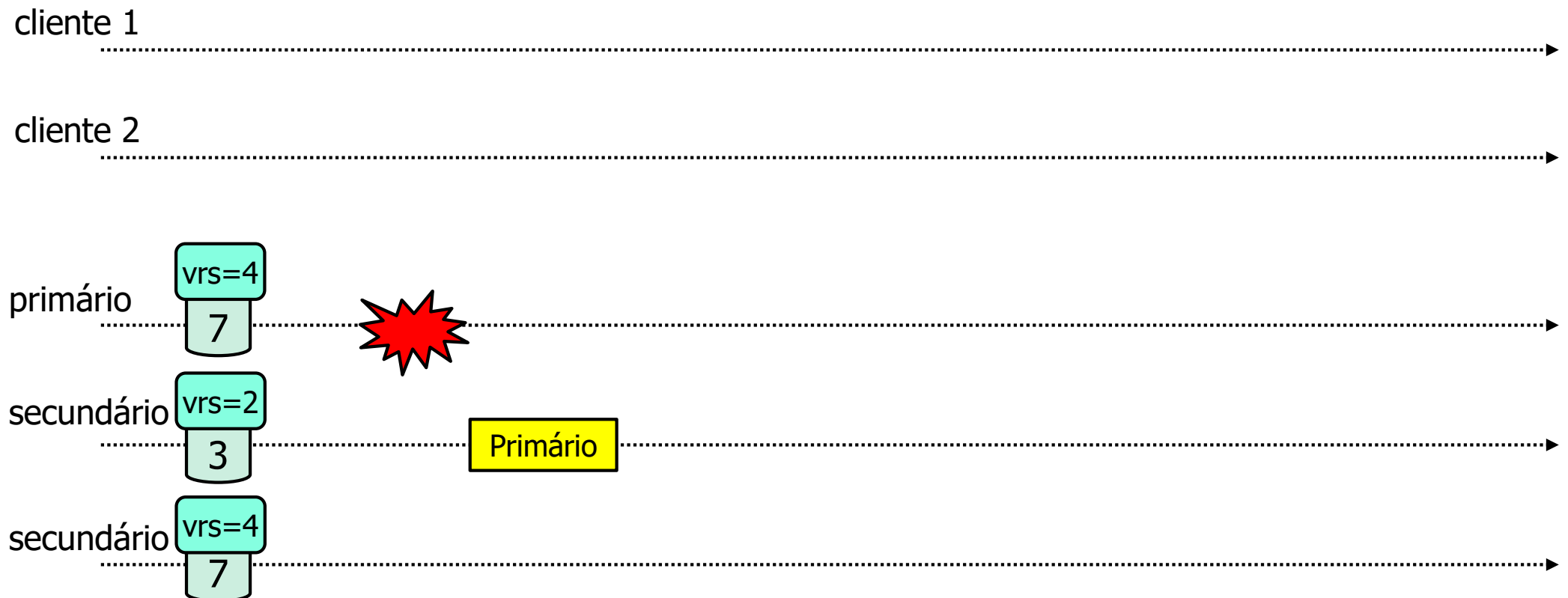
1. lista a diretoria e fica a observar alterações - se o seu nome for o menor, é o primário



PRIMÁRIO / SECUNDÁRIO : NOVO PRIMÁRIO

O que é que um nó faz quando descobre que há um novo primário?

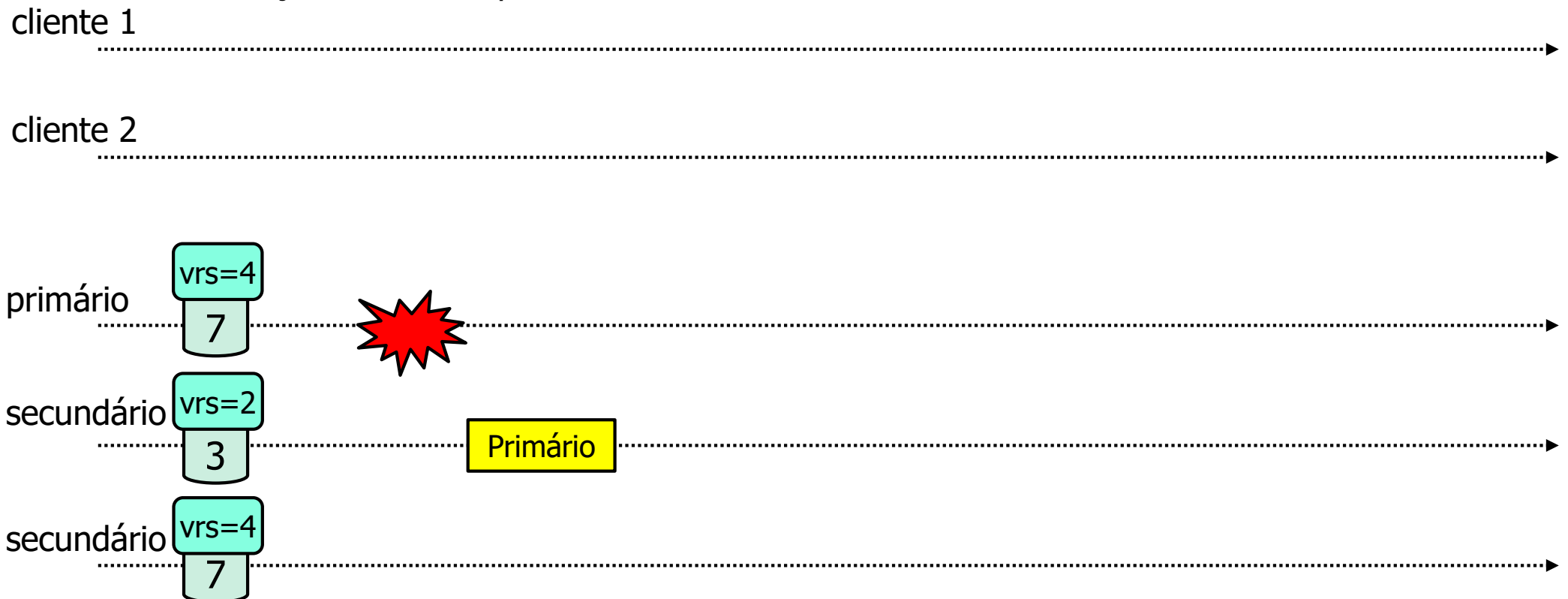
1. Deixa de receber mensagens do primário anterior (notem que o primário antigo pode não ter falhado ou ter recuperado)



PRIMÁRIO / SECUNDÁRIO : NOVO PRIMÁRIO

O que é que o novo primário deve fazer?

1. Deixa de aceitar operações do primário antigo. Ainda não executar operações como primário.
2. Contactar nós para obter operações/versões que não tenha
3. Começar a executar protocolo normal



REPLICAÇÃO DE MÁQUINA DE ESTADOS

Como garantir que todas as réplicas convergem para o mesmo estado ?

Existem muitos outros algoritmos além do primário/secundário.

No próximo capítulo veremos como implementar replicação de máquina de estados usando um sistema de comunicação indireta.

RESUMO

Introdução à replicação

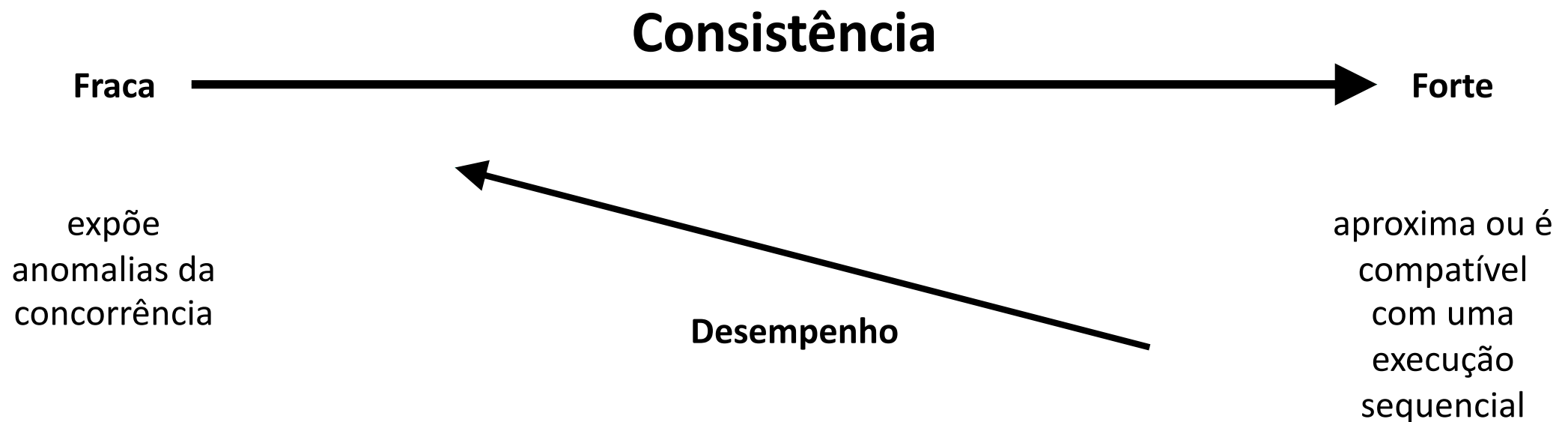
- Consistência forte. E.g.: primário-secundário
- **Consistência fraca.**

Caching

- Sistemas de ficheiros distribuídos
- Caching NFS
- Caching CIFS
- Caching Callback Promise

CONSISTÊNCIA

Qualidade que se refere à coerência dos dados face a atualizações concorrentes...(vs. uma execução puramente sequencial)



CONSISTÊNCIA "EVENTUAL" / FINAL

Cliente: os clientes podem **executar** as operações em **qualquer** servidor

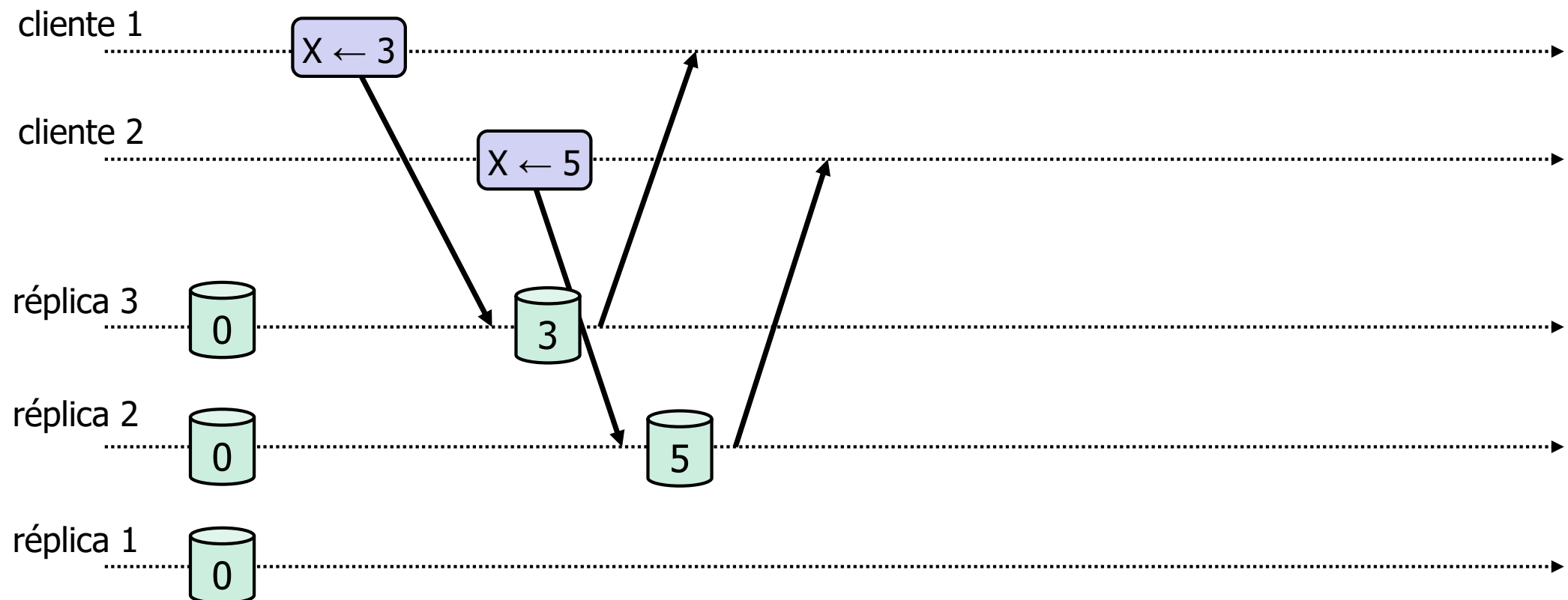
Réplicas: **todos** os servidores podem **receber** operações e executá-las **imediatamente**



CONSISTÊNCIA "EVENTUAL" / FINAL

Cliente: os clientes podem **executar** as operações em **qualquer** servidor

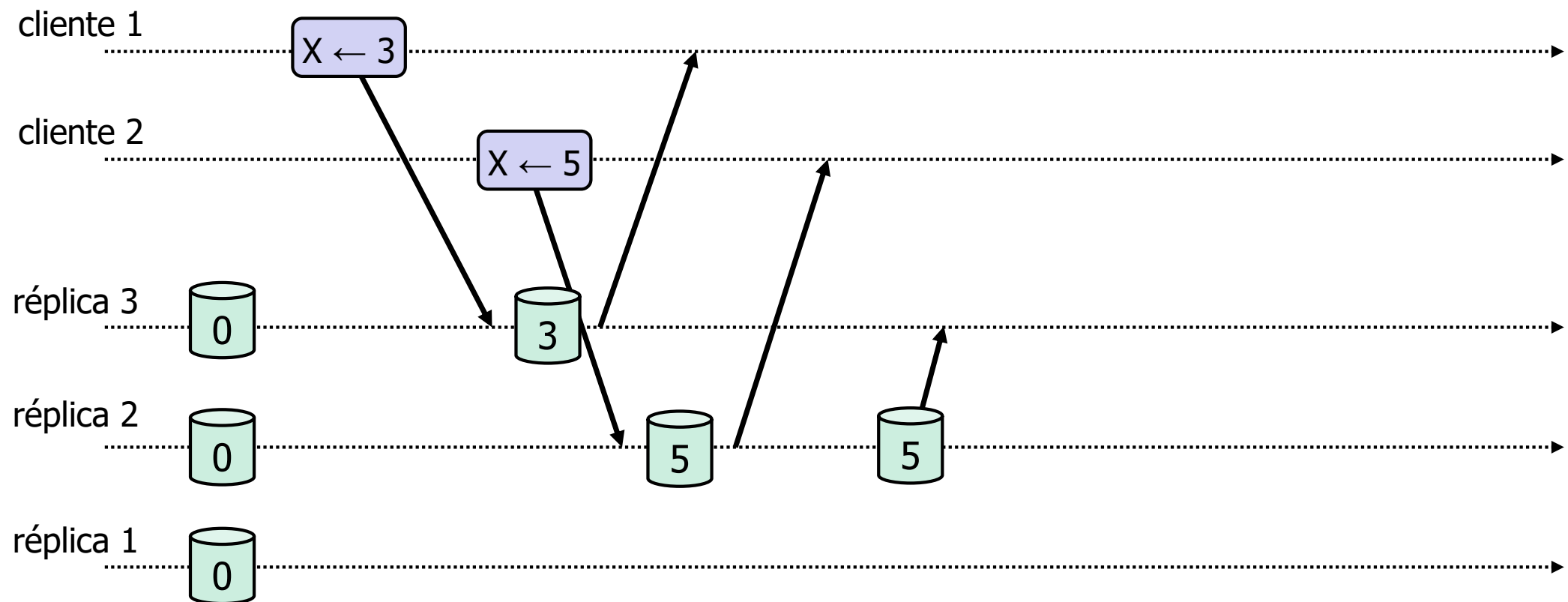
Réplicas: **todos** os servidores podem **receber** operações e executá-las **imediatamente**



CONSISTÊNCIA "EVENTUAL" / FINAL

Sincronização epidémica: réplicas comunicam periodicamente **trocando** as suas atualizações (podem enviar o estado ou a lista de operações)

- Como lidar com escritas concorrentes?
- **Possível solução:** última escrita ganha => ordem total sobre as escritas
- Como ordenar totalmente as operações?

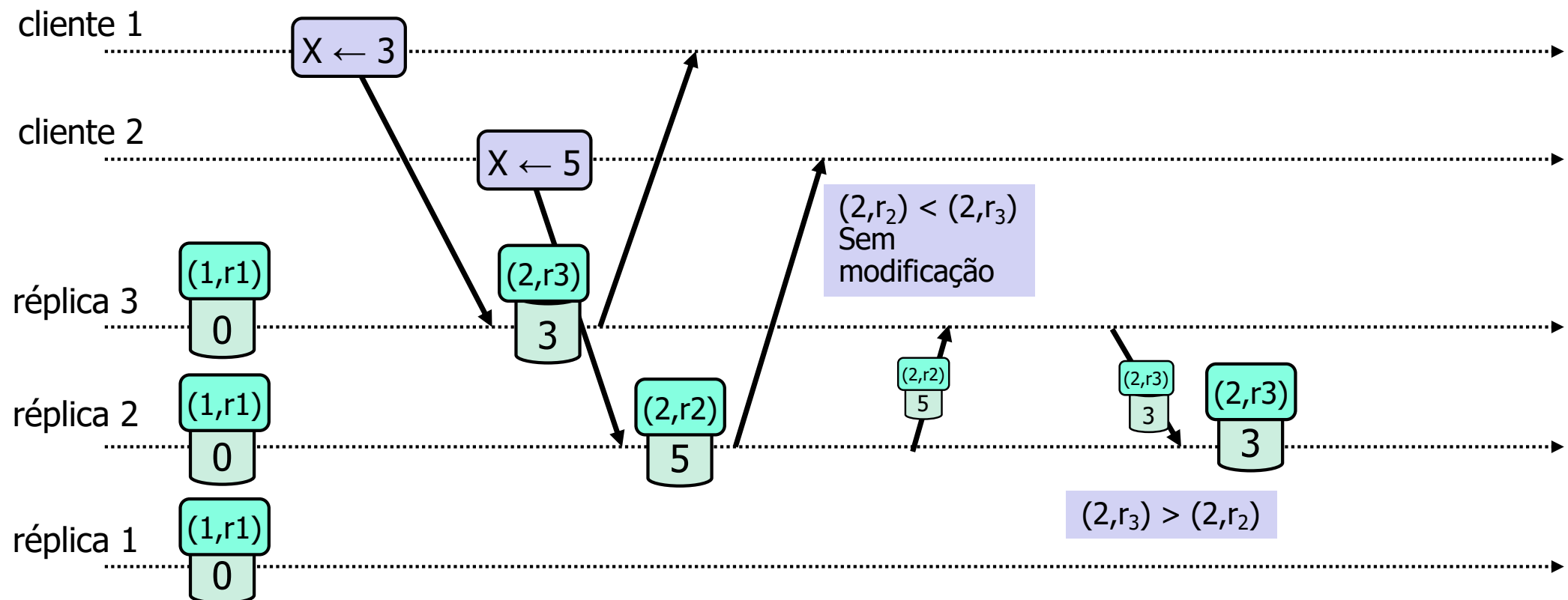


CONSISTÊNCIA "EVENTUAL" / FINAL

Como ordenar as operações? Uma hipótese...

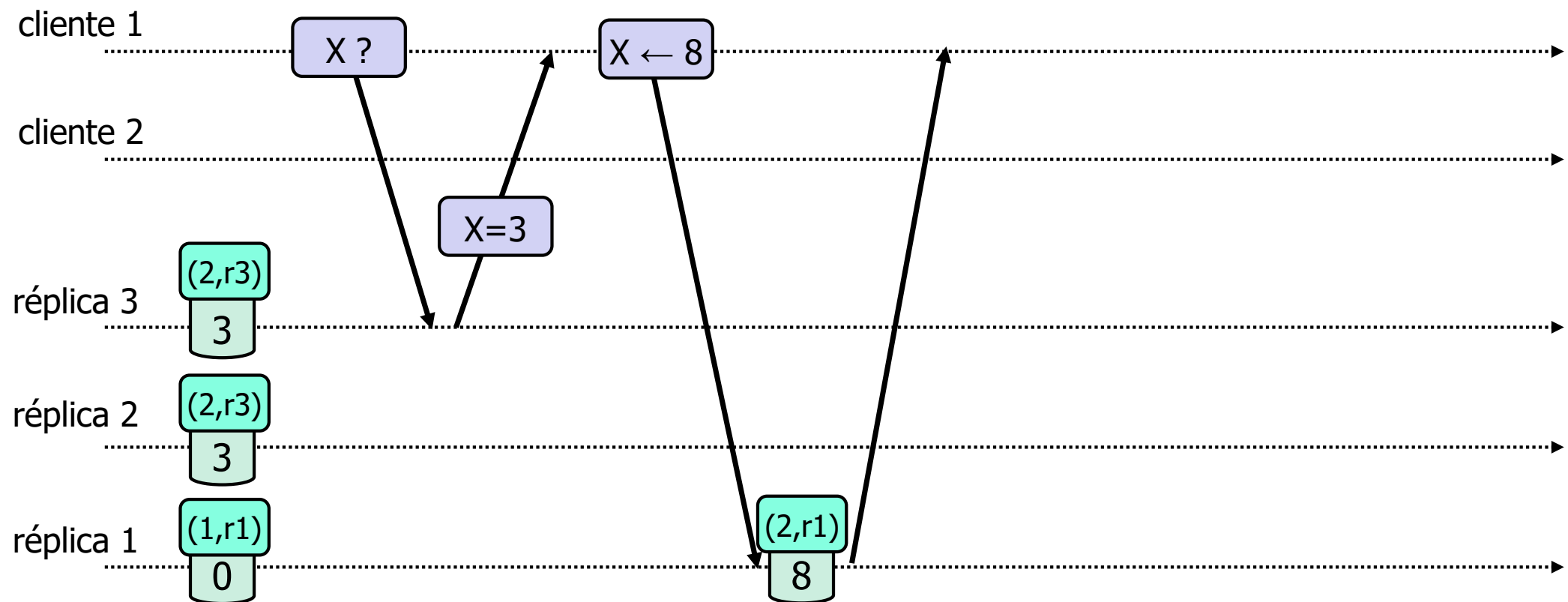
(relógio lógico de Lamport, id da réplica)

$(t_1, r_1) < (t_2, r_2)$ sse $t_1 < t_2$ OR $(t_1 = t_2 \text{ AND } r_1 < r_2)$



CONSISTÊNCIA "EVENTUAL" / FINAL

Problema?

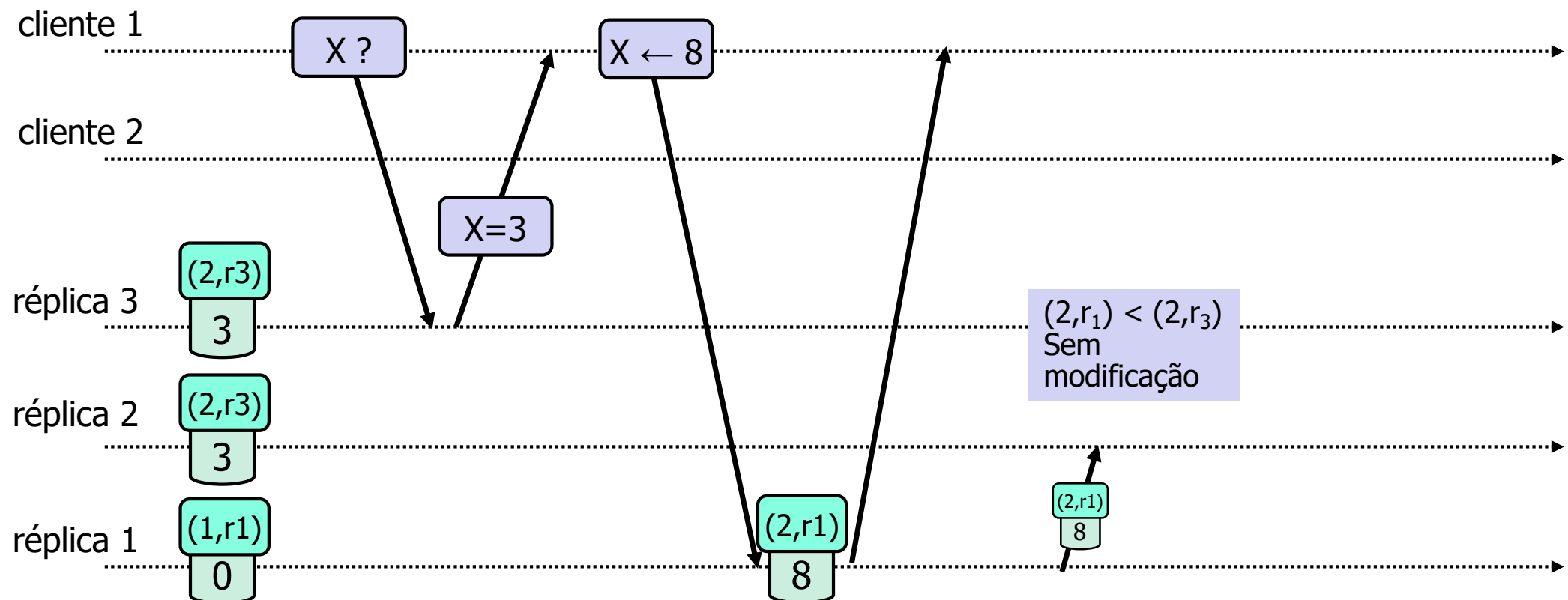


CONSISTÊNCIA "EVENTUAL" / FINAL

Problema?

Como $(2, r_1) < (2, r_3)$, a última escrita é preterida.

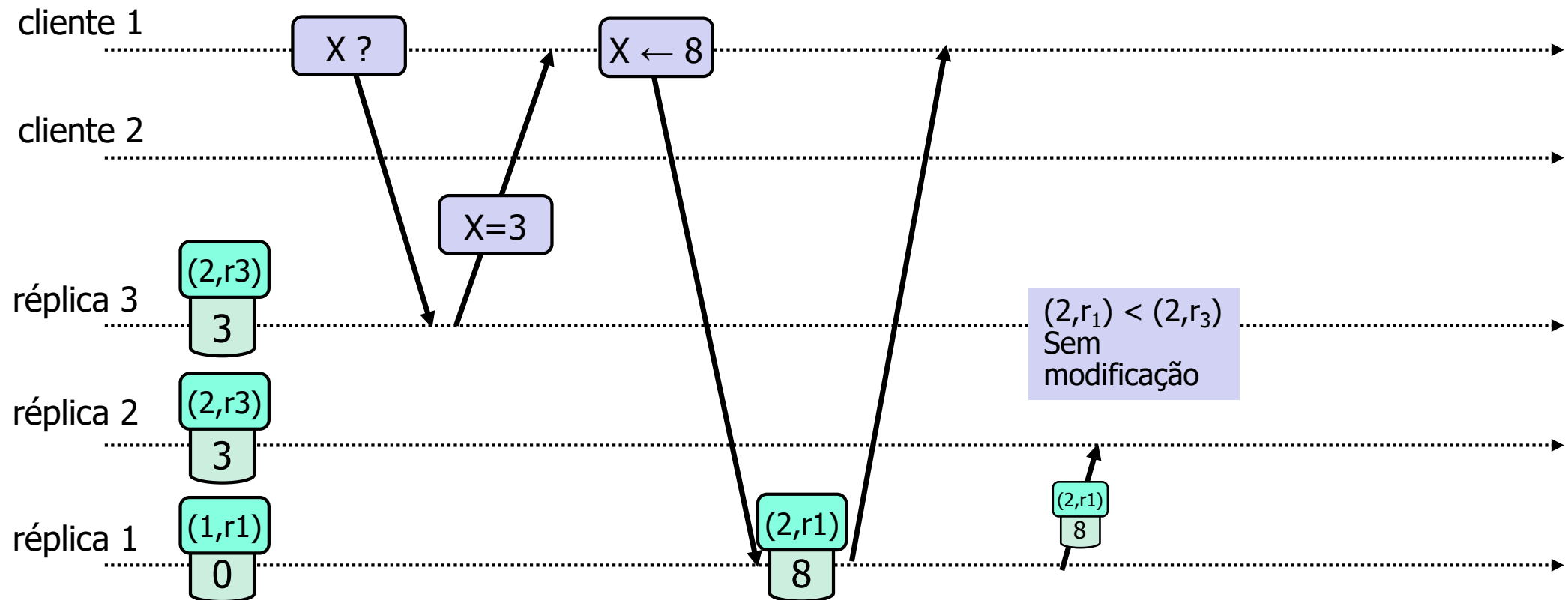
Dado que o cliente fez a escrita $X=8$ depois de ver o valor $X=3$, o resultado deveria ser $X=8$.



CONSISTÊNCIA "EVENTUAL" / FINAL

Propagar a causalidade é importante.

Solução?

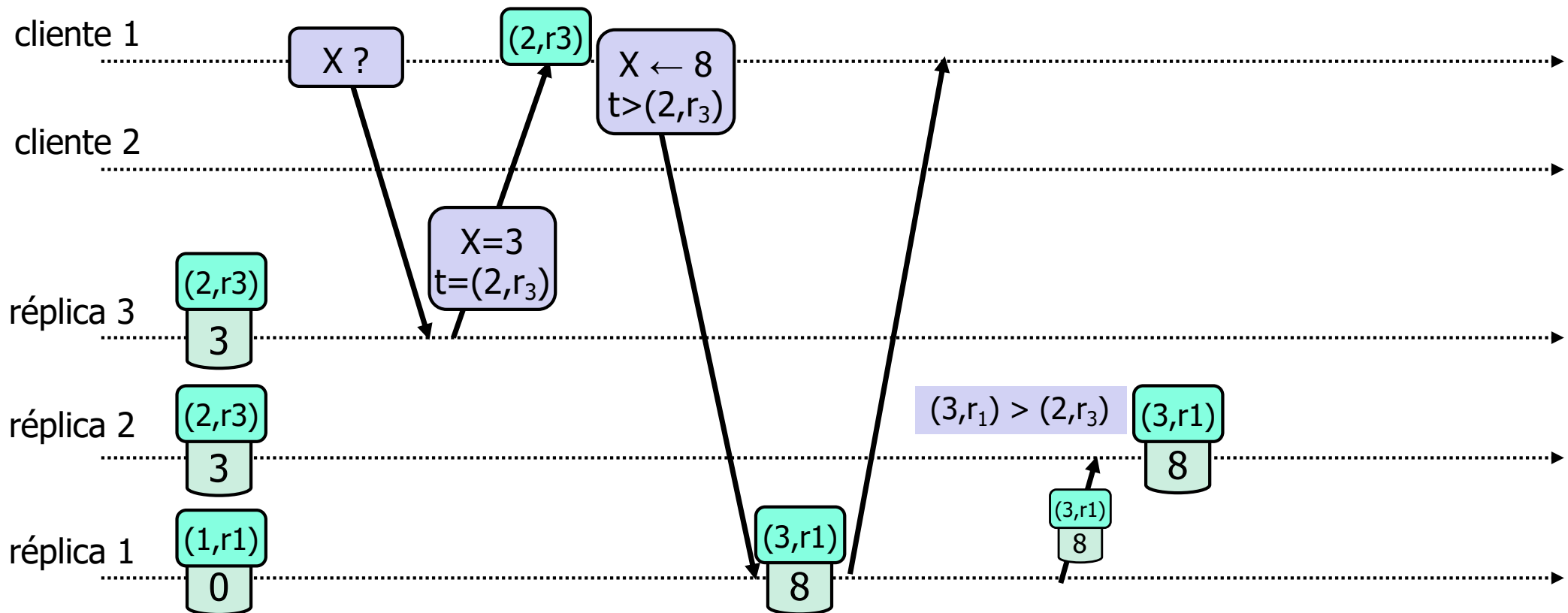


CONSISTÊNCIA "EVENTUAL" / FINAL

Propagar a causalidade é importante.

Solução?

Propagar relógios lógicos para o cliente.



CONSISTÊNCIA EVENTUAL

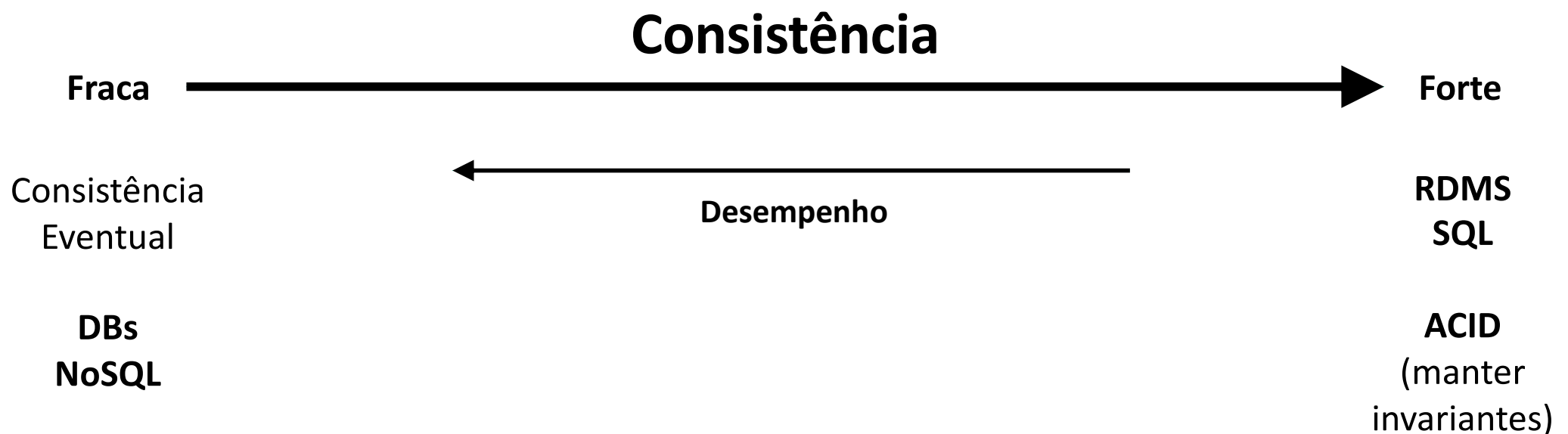
Replicação, cujo modelo de consistência entre réplicas promete **elevada disponibilidade** e que, **quando as atualizações terminarem**, as leituras acabarão por devolver **o mesmo valor**.

Ou seja, garante que **as réplicas irão convergir para um mesmo valor se não houver mais atualizações**.

Uma consequência é que **duas réplicas que tenham visto o mesmo conjunto de atualizações têm o mesmo estado**.

CONSISTÊNCIA

Qualidade que se refere à coerência dos dados face a atualizações concorrentes...(vs. uma execução puramente sequencial)



AMAZON DYNAMO (2007)

Sistema de bases de dados NoSQL

Replicação geográfica com elevada disponibilidade e desempenho

Operações que atualizam o estado podem executar sem ver os efeitos uma da outra.

“Eventually”, todas as operações são propagadas a todas as réplicas.

DYNAMO: INTERFACE

get(key) → returns <list of values,context>

- Retorna lista de escritas mais recentes que não viram os efeitos uma da outra (de forma a nenhuma se perder)
- **Context** descreve o conjunto de escritas que são reflectidas na lista de valores retornada

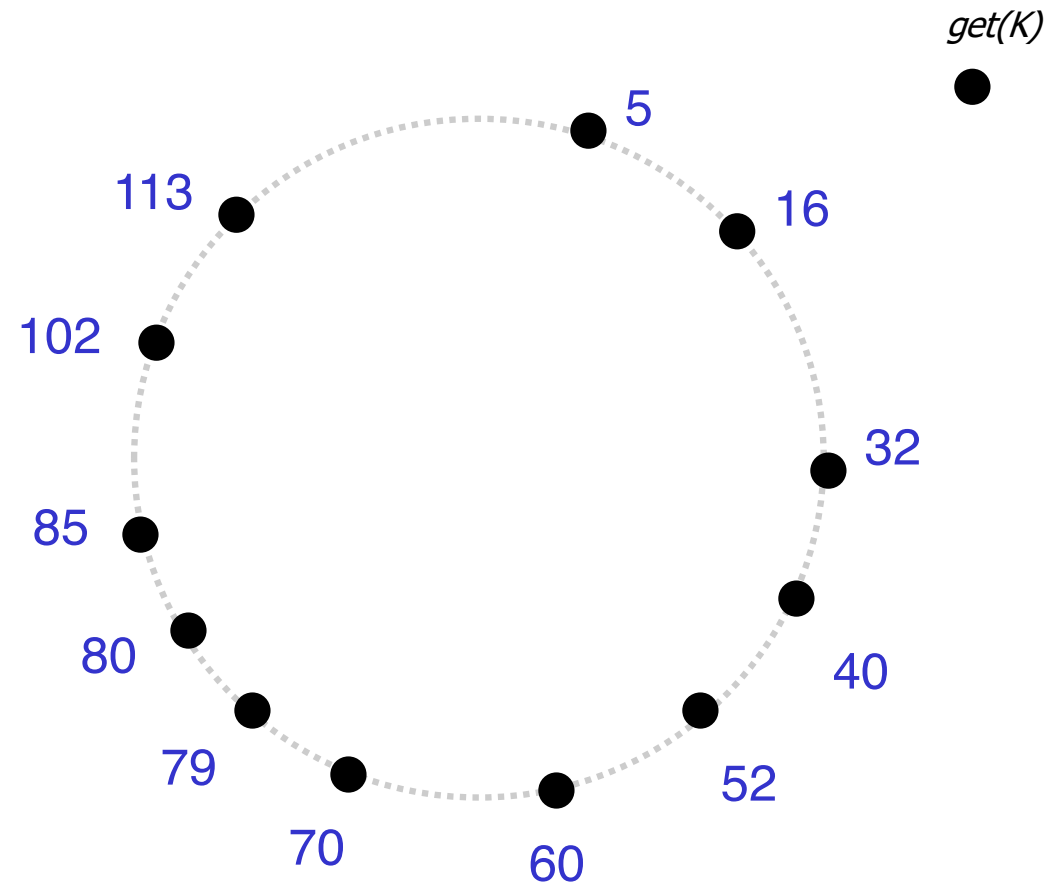
put(key,value,context) → returns "ack"

- Escreve o novo valor "value"
- Cliente deve passar o contexto associado ao get mais recente no qual a escrita se baseia

DYNAMO: IMPLEMENTAÇÃO

Nós do sistema organizados numa DHT one-hop, i.e., uma DHT em que um nó conhece todos os outros.

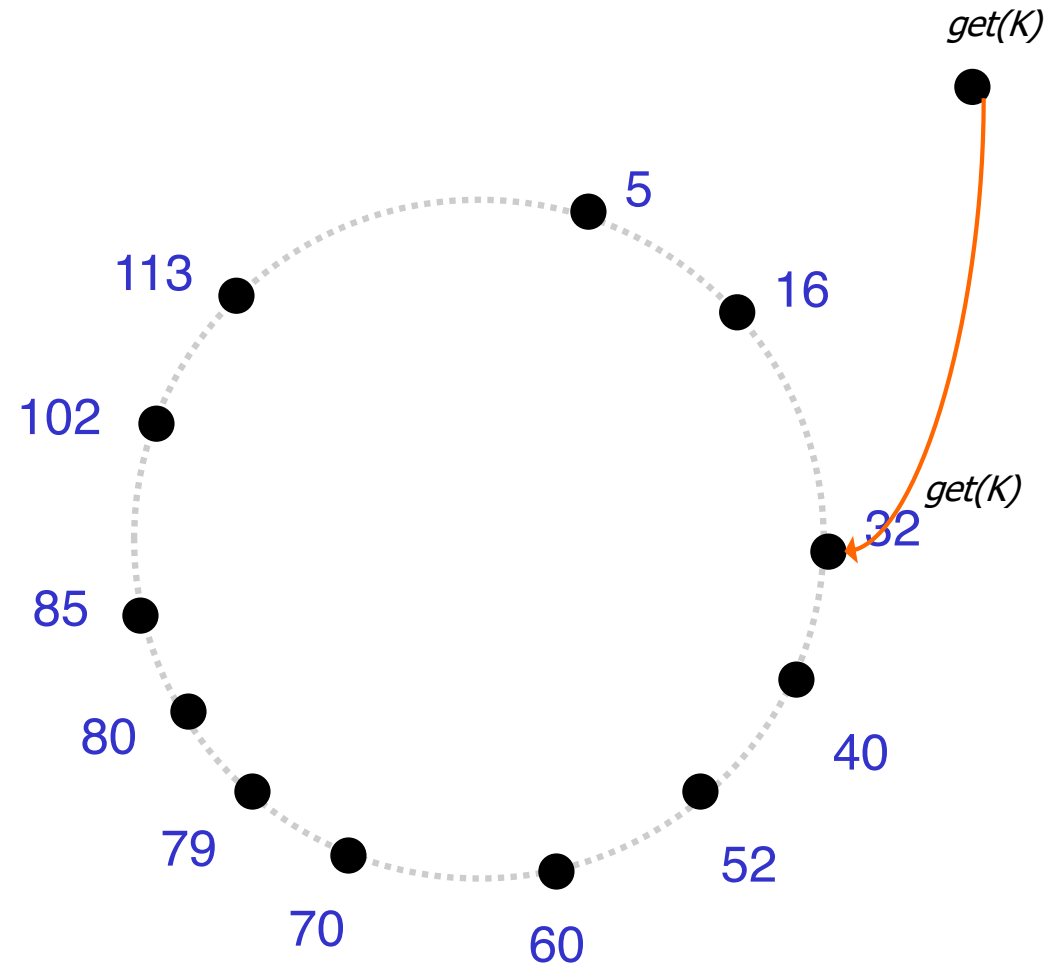
Quóruns de leitura e escrita: subconjuntos de R e W réplicas, com $R+W < N$ (podem não se intersectar)



DYNAMO: IMPLEMENTAÇÃO

Operações executam-se numa única fase:

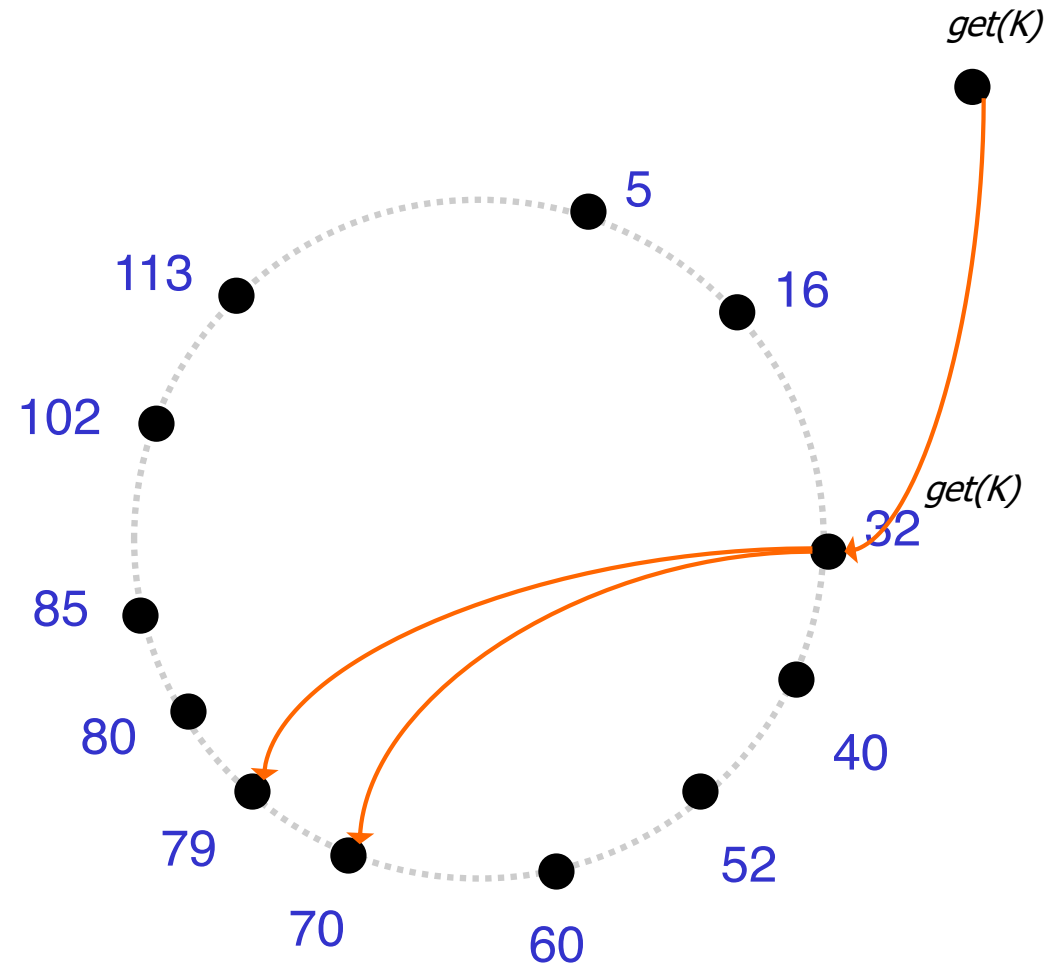
1. Cliente contacta uma das réplicas escolhida aleatoriamente



DYNAMO: IMPLEMENTAÇÃO

Operações executam-se numa única fase:

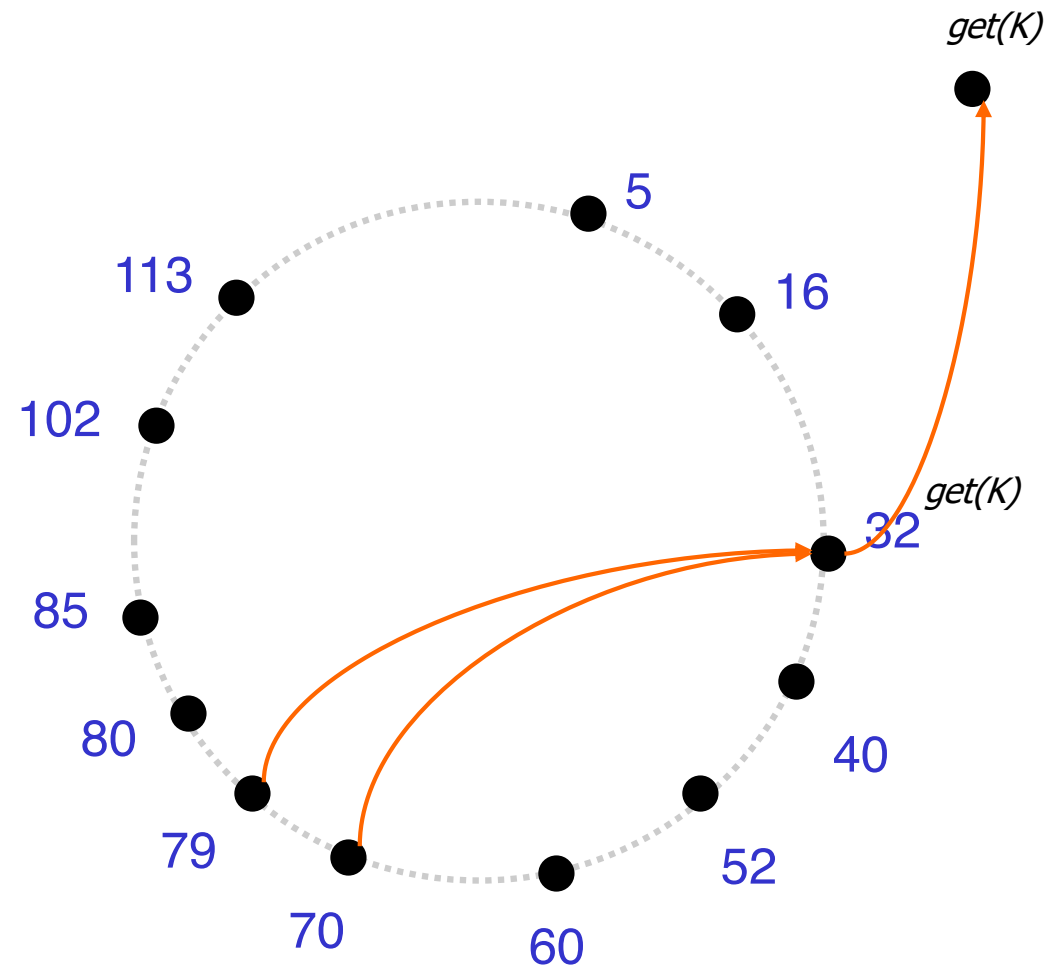
1. Cliente contacta uma das réplicas escolhida aleatoriamente
2. Réplica propaga pedido de get/put para N réplicas



DYNAMO: IMPLEMENTAÇÃO

Operações executam-se numa única fase:

1. Cliente contacta uma das réplicas escolhida aleatoriamente
2. Réplica propaga pedido de get/put para N réplicas
3. Espera resposta de R/W réplicas antes de retornar ao cliente
4. No caso do get, retorna o(s) valor(es) mais recente(s) – mais do que um no caso de escritas concorrentes



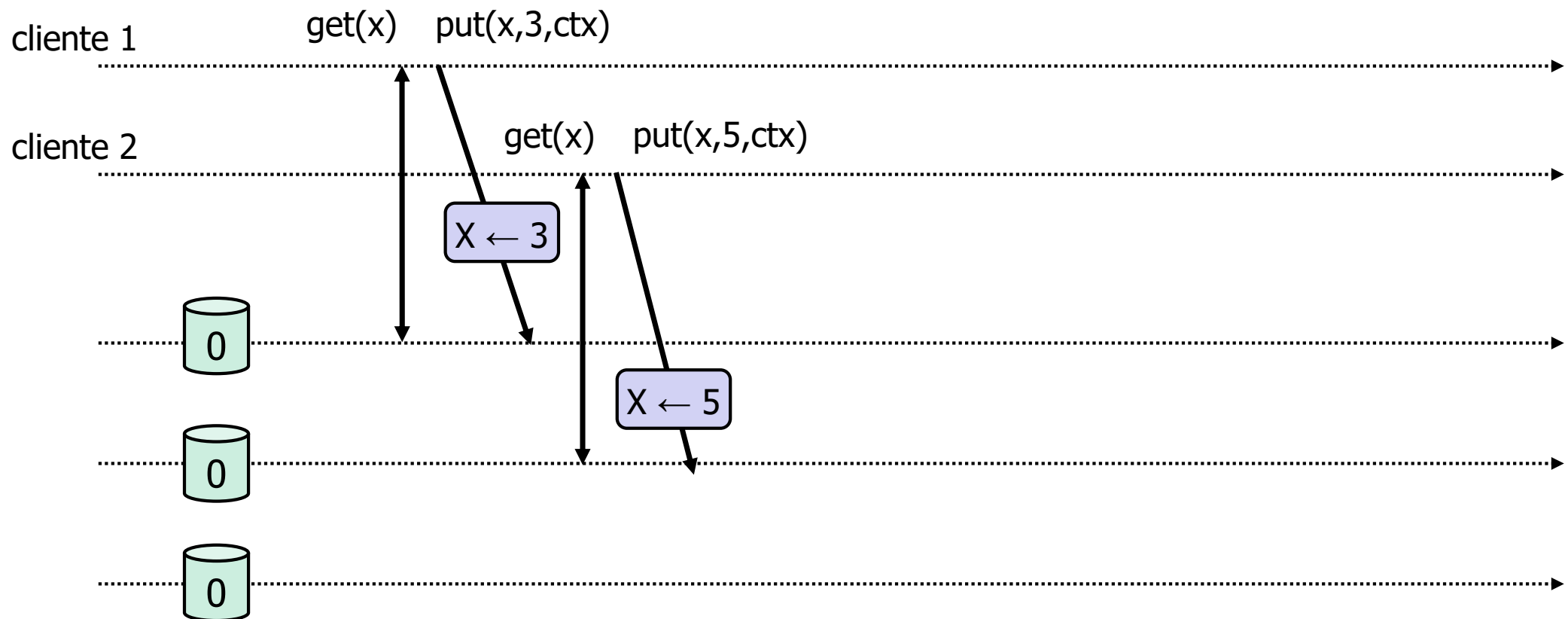
DYNAMO: ESCRITAS CONCORRENTES

No Dynamo podem ocorrer escritas concorrentes porque:

- Vários clientes podem escrever ao mesmo tempo

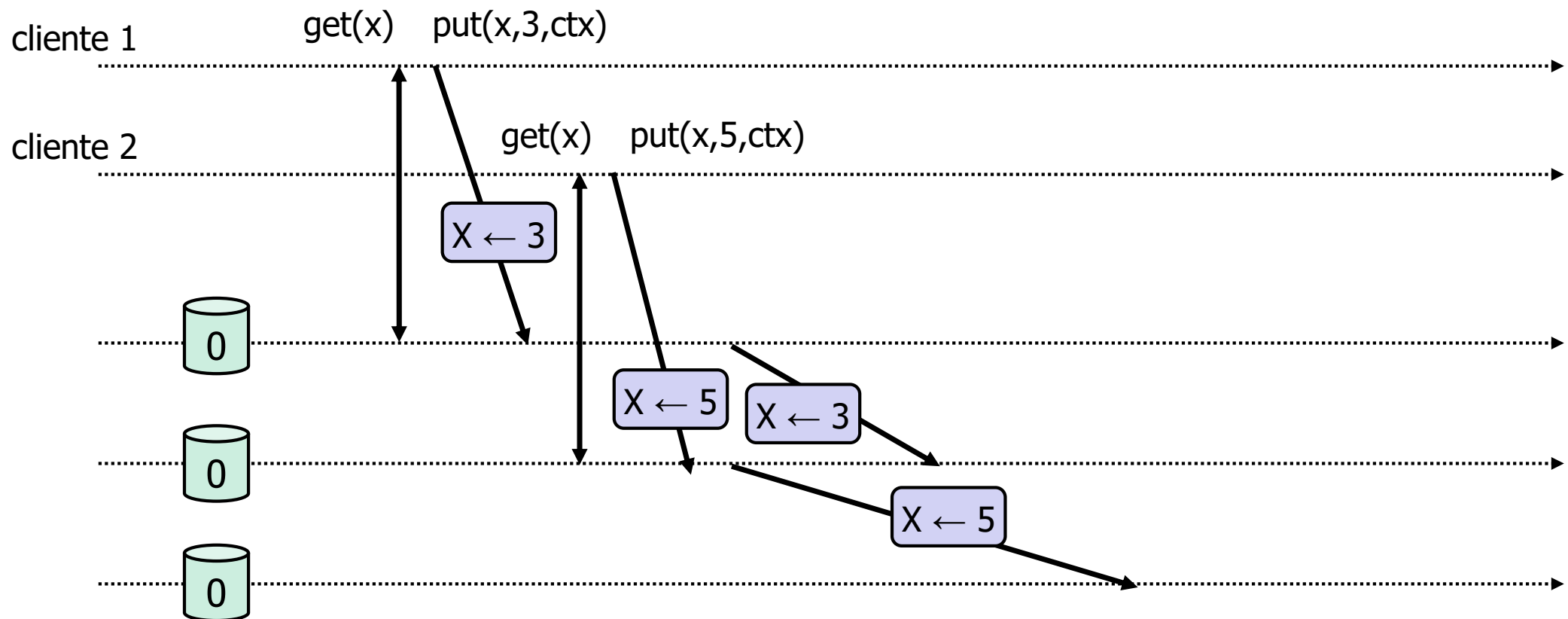
DYNAMO: ESCRITAS CONCORRENTES

Dois clientes fazem escritas concorrentes, i.e., sem conhecerem a escrita do outro.



DYNAMO: ESCRITAS CONCORRENTES

Dois clientes fazem escritas concorrentes, i.e., sem conhecerem a escrita do outro.



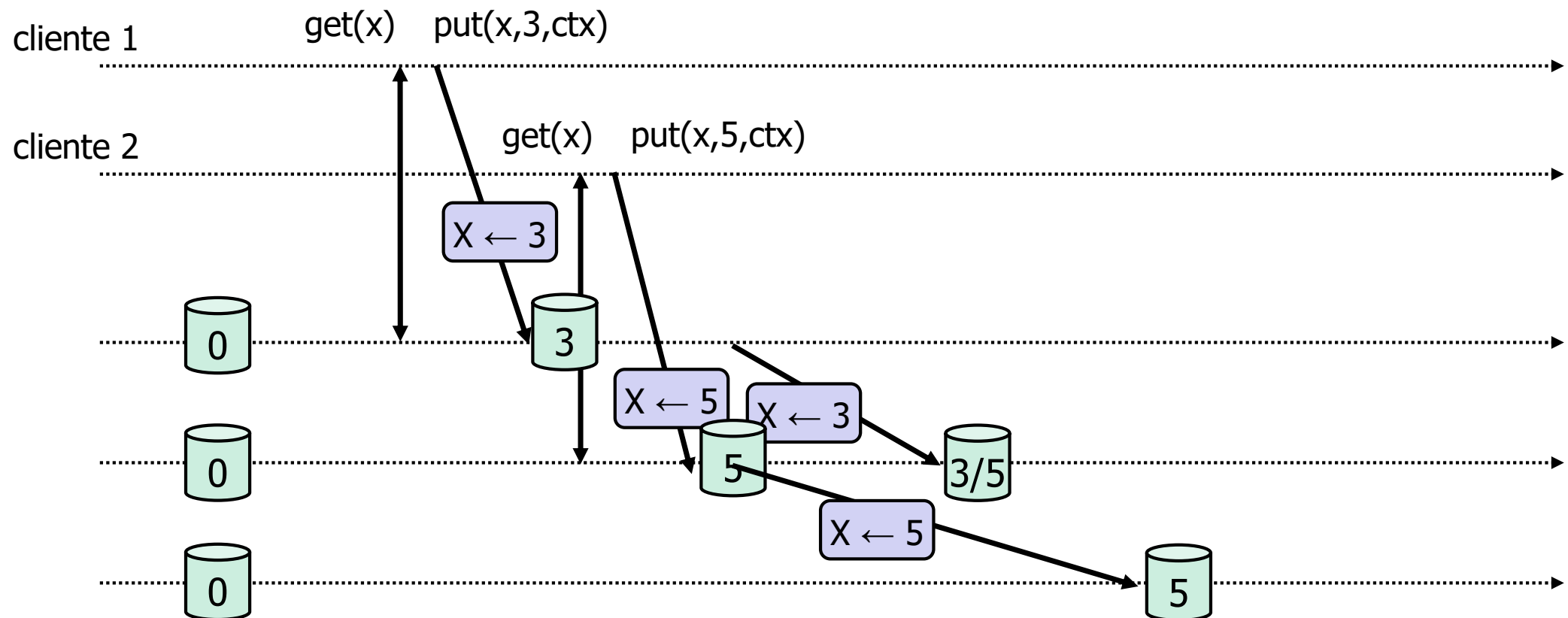
DYNAMO: ESCRITAS CONCORRENTES

No Dynamo podem ocorrer escritas concorrentes porque:

- Vários clientes podem escrever ao mesmo tempo
- Sistema permite que se escreve em servidores que não replicam a chave (sloppy quorums)

DYNAMO: COMO LIDAR COM ESCRITAS CONCORRENTES?

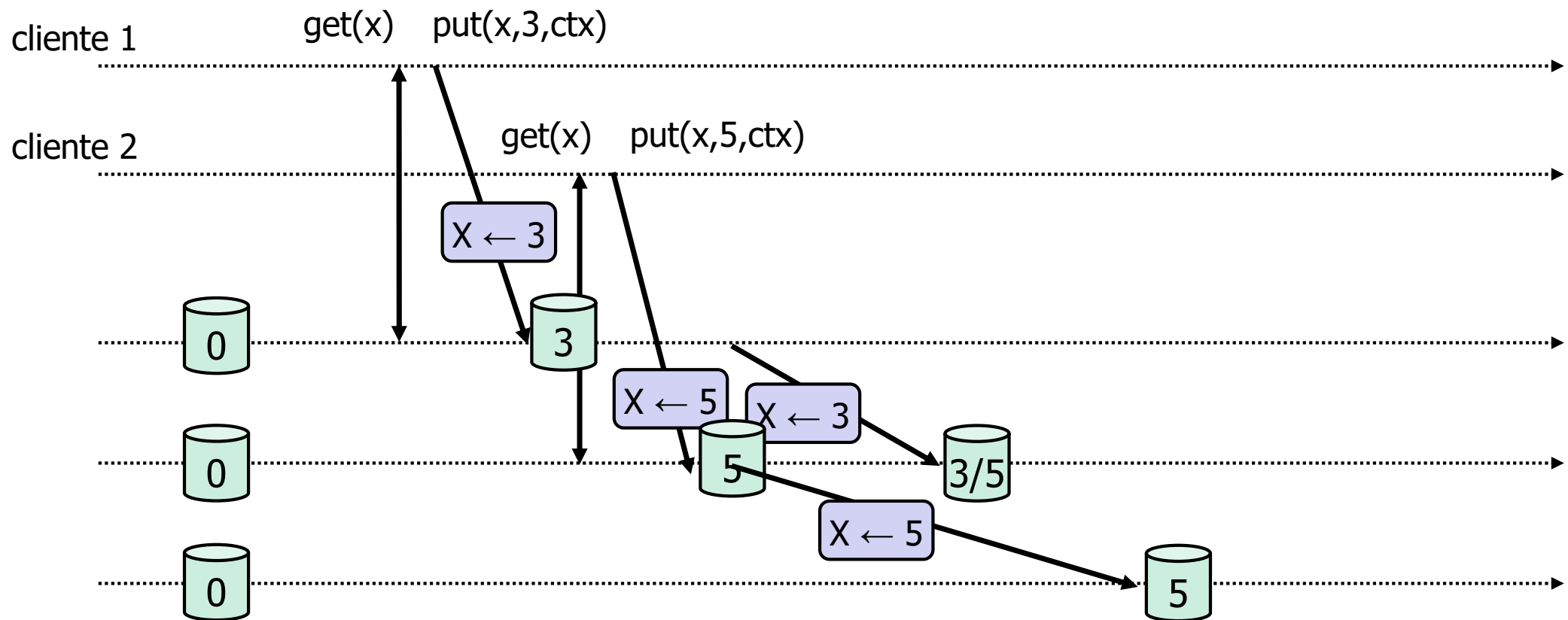
Sistemas mantêm as várias escritas concorrentes e devolve-as ao cliente na leitura.



DYNAMO: ESCRITAS CONCORRENTES

Como detetar escritas concorrentes?

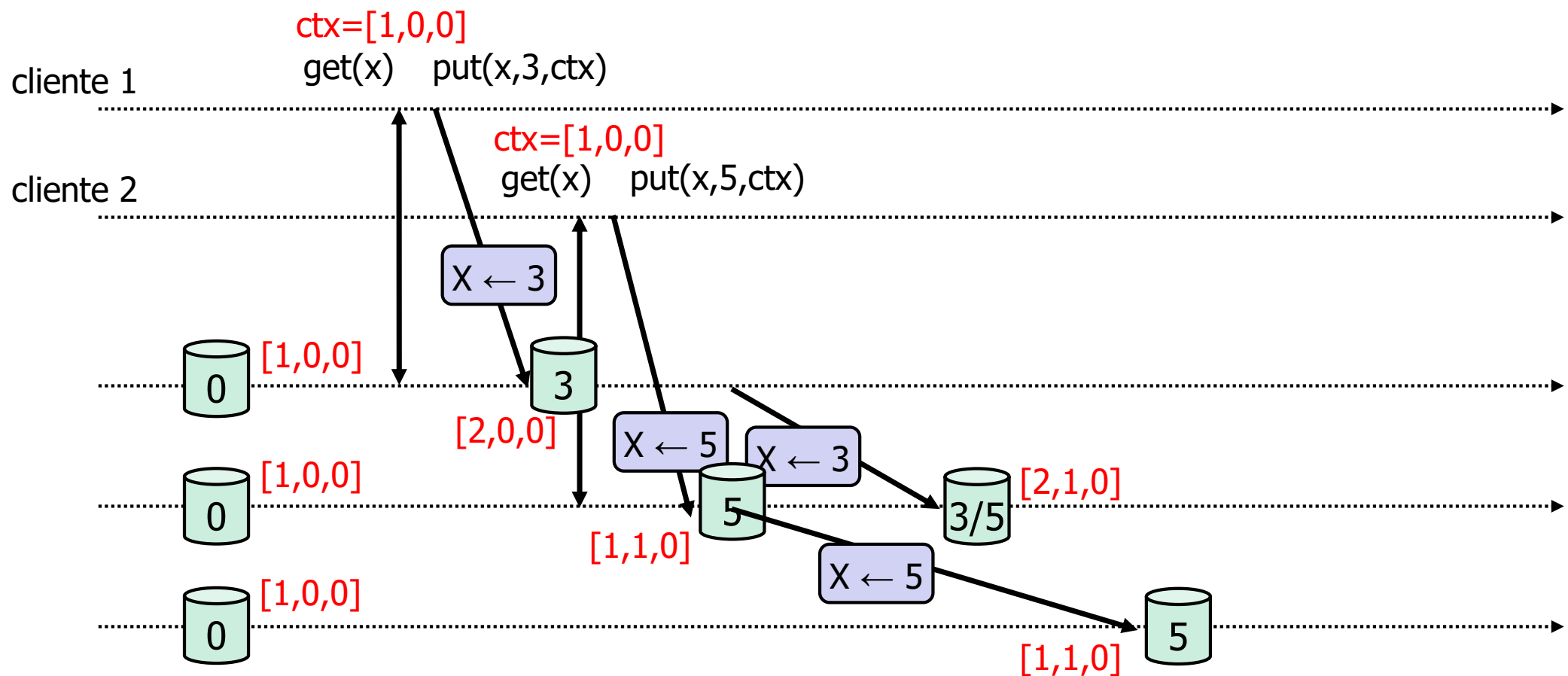
Usando vetores versão.



DYNAMO: ESCRITAS CONCORRENTES

Como detetar escritas concorrentes?

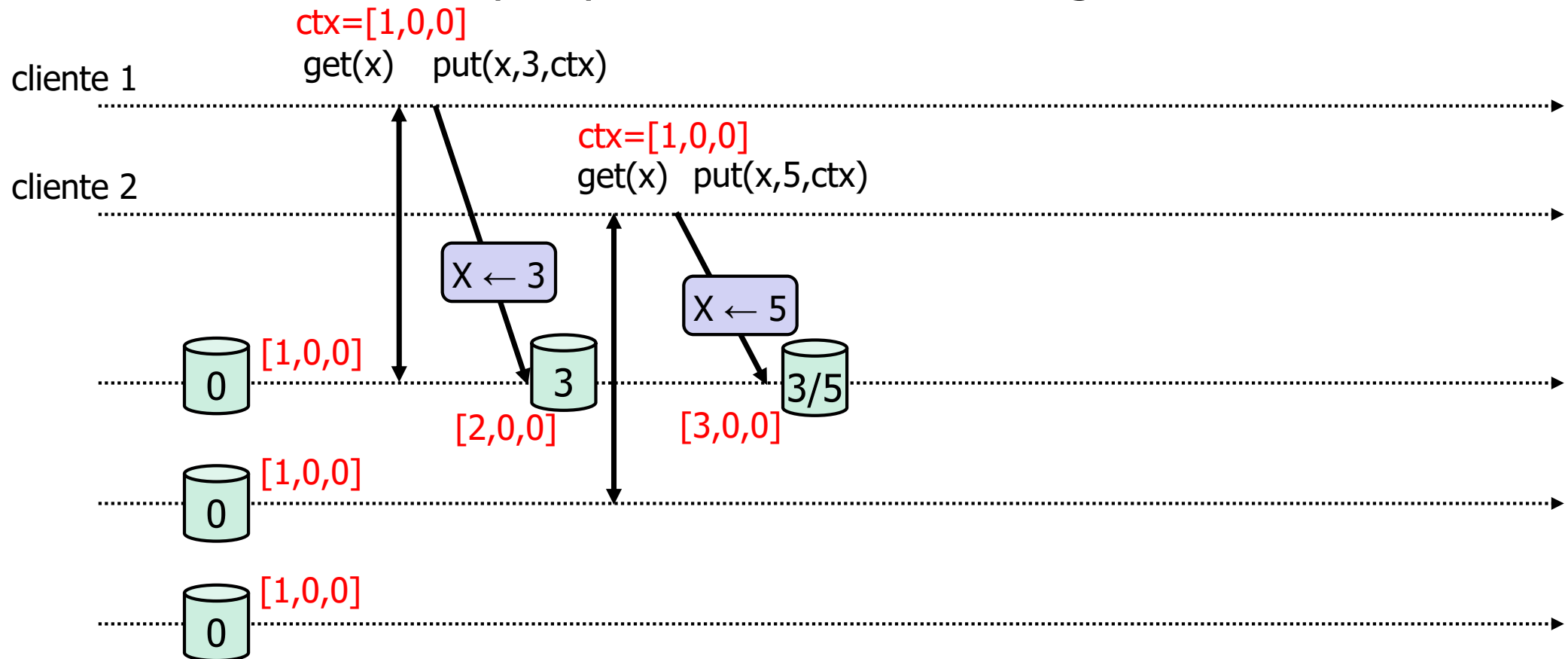
Usando vetores versão.



DYNAMO: ESCRITAS CONCORRENTES (ALTERNATIVA 2)

Como detetar escritas concorrentes?

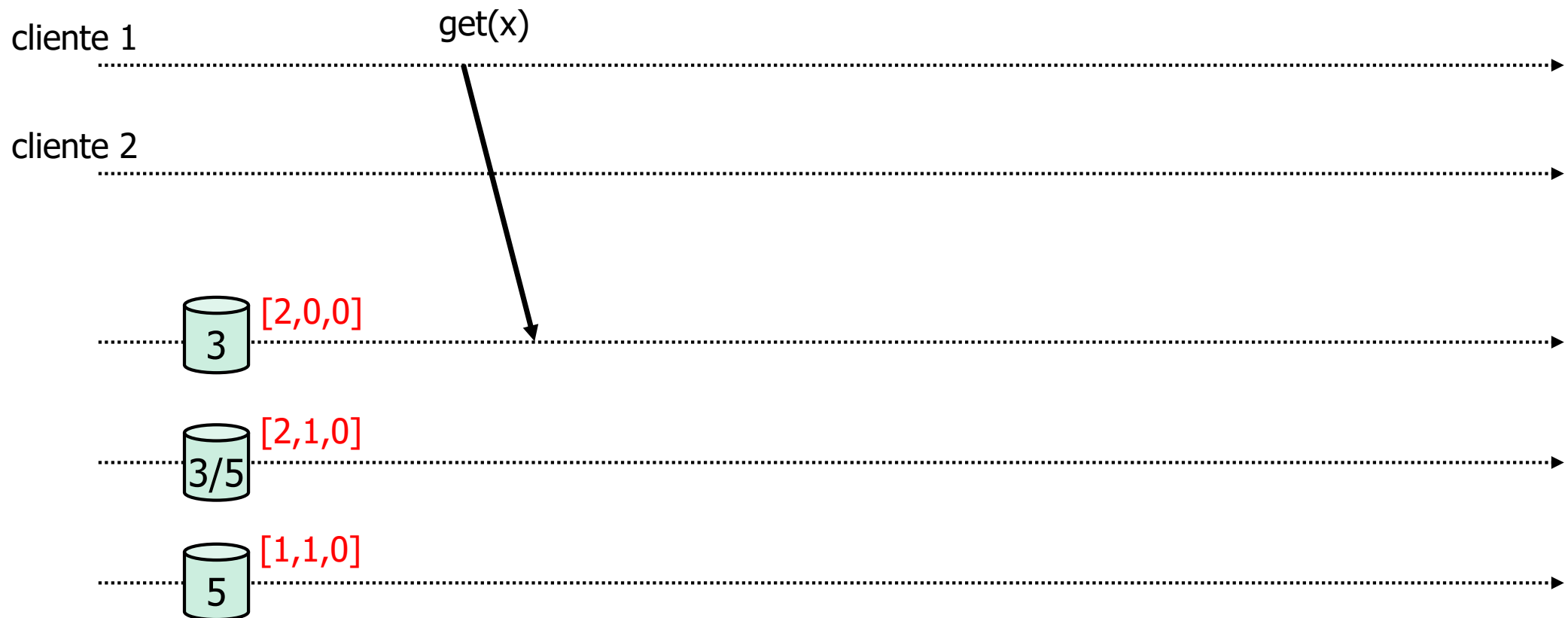
Quando se recebe uma escrita, se contexto menor que contexto atual é porque houve escrita – e.g. $[1,0,0] < [2,0,0]$



DYNAMO: ESCRITAS CONCORRENTES

Como é que a leitura devolve escritas concorrentes?

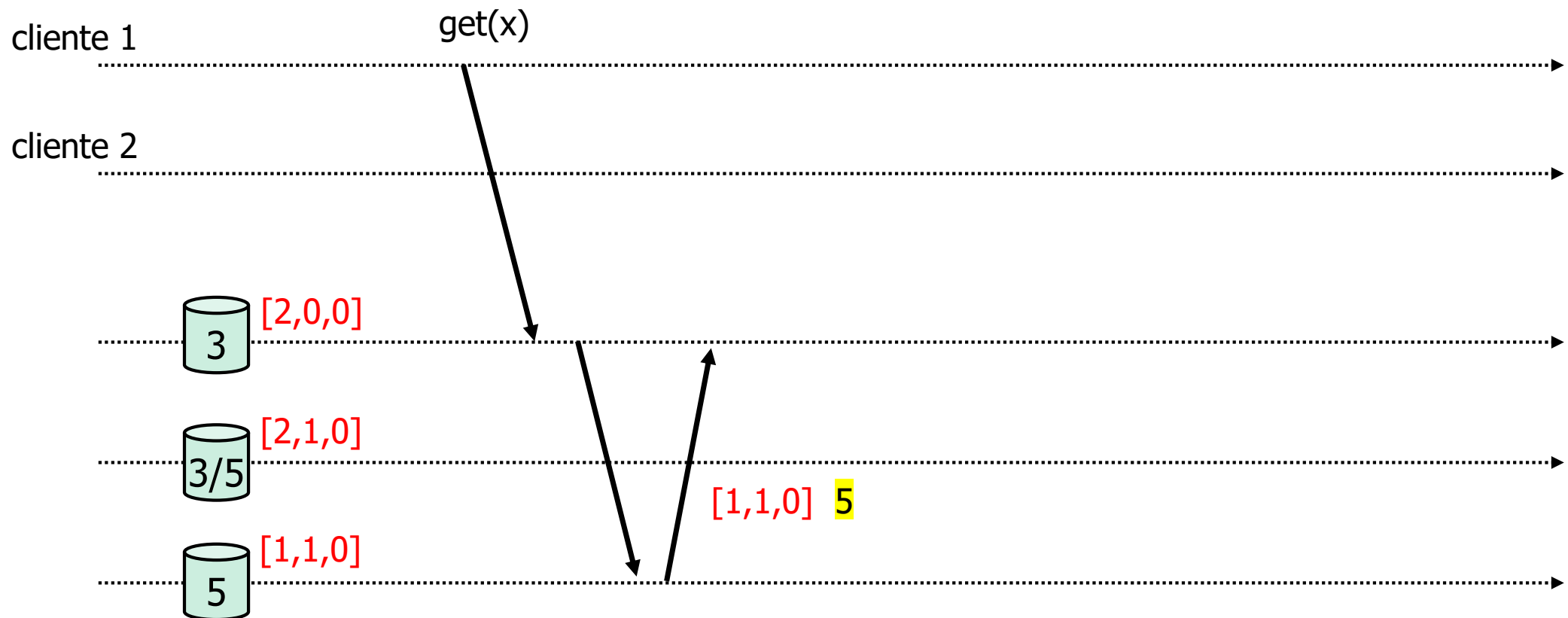
1. Cliente contacta réplica



DYNAMO: ESCRITAS CONCORRENTES

Como é que a leitura devolve escritas concorrentes?

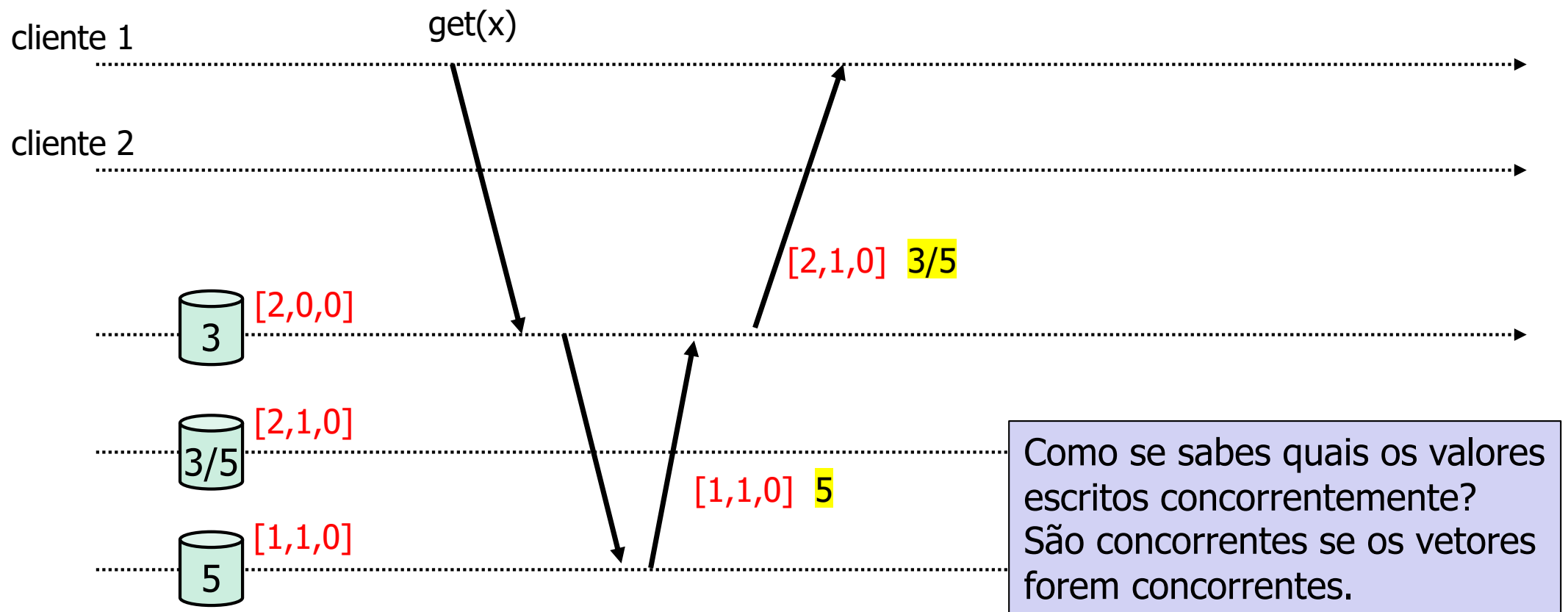
2. Réplica contacta R-1 réplicas



DYNAMO: ESCRITAS CONCORRENTES

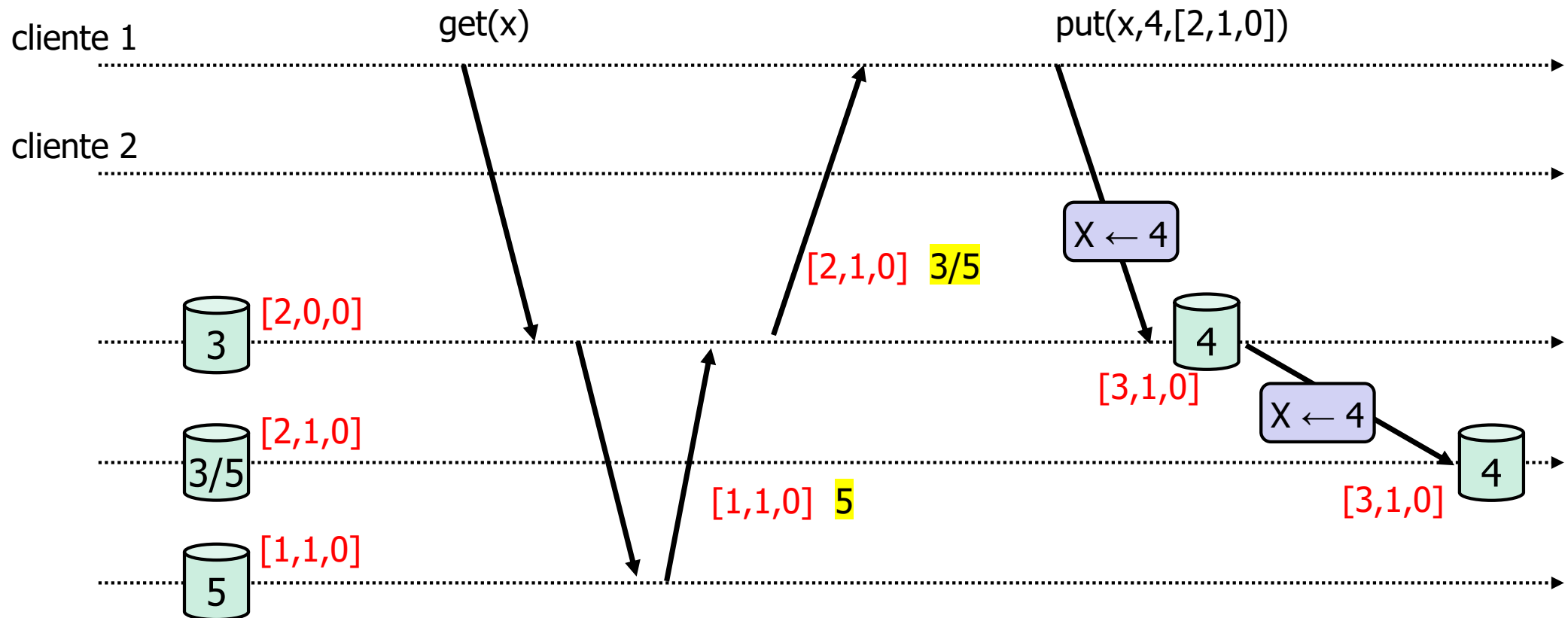
Como é que a leitura devolve escritas concorrentes?

3. Réplica devolve lista de valores escritos concorrentes e vetor versão que unifica vetor versão das réplicas contactadas.



DYNAMO: RESOLUÇÃO DE CONFLITOS

Cliente pode escrever valor que resolve conflito.



PROBLEMA: RECONCILIAR DIVERGÊNCIA

Problema: como lidar com escritas que executaram de forma concorrente?

- E.g. dado um carrinho de compras com {batatas}, o que acontece se um cliente escrever o novo estado: {batatas,cebolas} e outro cliente escrever concorrentemente: {nabos}

PROBLEMA: RECONCILIAR DIVERGÊNCIA

Problema: como lidar com escritas que executaram de forma concorrente?

- E.g. dado um carrinho de compras com {batatas}, o que acontece se um cliente escrever o novo estado: {batatas,cebolas} e outro cliente escrever concorrentemente: {nabos}

Dynamo: cliente responsável por resolver conflito.

Artigo do Dynamo sugere que se faça a união dos carrinhos de compras -> {batatas,cebolas,nabos}

PROBLEMA: RECONCILIAR DIVERGÊNCIA

Problema: como lidar com escritas que executaram de forma concorrente?

- E.g. dado um carrinho de compras com {batatas}, o que acontece se um cliente escrever o novo estado: {batatas,cebolas} e outro cliente escrever concorrentemente: {nabos}

Cassandra (clone Dynamo open-source): “last-writer-wins”

Resultado seria {batatas,cebolas} ou {nabos}, dependendo da estampilha da escrita.

PROBLEMA: RECONCILIAR DIVERGÊNCIA

Problema: como lidar com escritas que executaram de forma concorrente?

- E.g. dado um carrinho de compras com {batatas}, o que acontece se um cliente escrever o novo estado: {batatas,cebolas} e outro cliente escrever concorrentemente: {nabos}

CRDTs (conflict-free replicated data types): tipos de dados com política que unifica atualizações e inclui política de resolução de conflitos pré-definida

Resultado seria {cebolas,nabos}, no pressuposto que o primeiro cliente adicionou cebolas e o segundo retirou as batatas.

SUMÁRIO

Introdução à replicação

- Primário-secundário
- Multi-master

Caching

- Sistemas de ficheiros distribuídos
- Caching NFS
- Caching CIFS
- Caching Callback Promise

CACHING

O que é *Caching* ou uma *Cache*?

"... Um lugar para armazenar algo temporariamente num ambiente de computação..."

É uma forma de replicação?

Sim..., mas as réplicas são (muito) mais numerosas, criadas a pedido e efémeras.

CACHING

Para que serve?

Disponibilidade?

- Em casos relativamente limitados, sim...
- e.g., Mascarar a desconexão em dispositivos móveis.

Desempenho?

- Definitivamente! Reduz a latência dos acessos, melhora o débito global de um sistema.

CACHING

Problemas fundamentais?

A **semântica tradicional centralizada** consiste em uma leitura ver sempre o resultado da última escrita (serialização dos acessos).

Num **sistema distribuído**, o problema complica-se devido à existência de caches em múltiplas máquinas.

A semântica dum mecanismo de gestão das caches depende do que se faz nas operações de leitura e escrita:

- Associado às **escritas** está a propagação das modificações para o servidor (e sua visualização por outros clientes).
- Associada às **leituras** está a verificação da coerência da cache em relação ao(s) servidor(es).

CASO DE ESTUDO - CACHING

Sistemas de Ficheiros Distribuídos

MOTIVAÇÃO PARA UTILIZAÇÃO DE SISTEMAS DE FICHEIROS DISTRIBUÍDOS

Ubiquidade dos acessos - acessível de diferentes dispositivos, em qualquer lugar

Partilha de dados por múltiplos utilizadores

Melhor qualidade de serviço do que a disponível localmente

- Melhor tolerância a falhas
- Maior disponibilidade
- Maior dimensão do espaço disponível
- Desempenho comparável ou superior (escritas em particular)

SISTEMAS DE FICHEIROS DISTRIBUÍDOS

Um sistema de **gestão de ficheiros distribuídos** fornece um serviço de acesso a ficheiros semelhante ao de um sistema de ficheiros normal, mas estendido a um conjunto de máquinas ligadas em rede.

INTEGRAÇÃO NO SISTEMA LOCAL

Os ficheiros dum sistema distribuído de ficheiros devem ser disponibilizados localmente numa máquina.

Problema: Qual a granularidade da partilha?

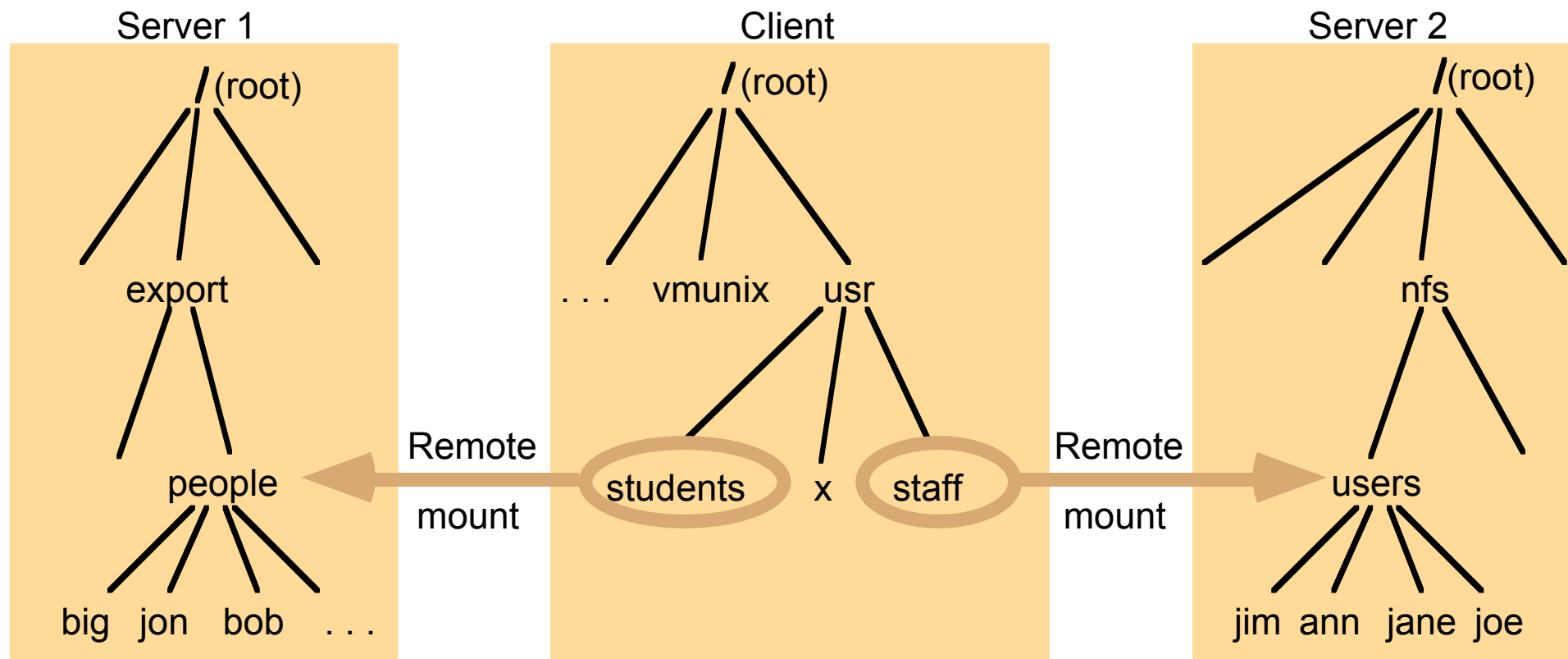
Solução típica: subárvore do sistema de ficheiros

Problema: Como providenciar acesso aos ficheiros?

Solução: integrar a subárvore na hierarquia local

- Windows: definindo uma nova *drive*
- Unix-like: através do mecanismo de *mount* ("*pendura*" um sistema de ficheiros como uma sub-arvore de outro)

EXEMPLO: MOUNT REMOTO E DESIGNAÇÃO NO NFS



Cada servidor exporta um conjunto de diretorias.

O cliente pode **montar** (colar/enxertar) uma diretoria remota (caso tenha permissões) numa diretoria local (ex.: `/usr/students` no cliente monta a árvore

`/export/people` no Server 1)

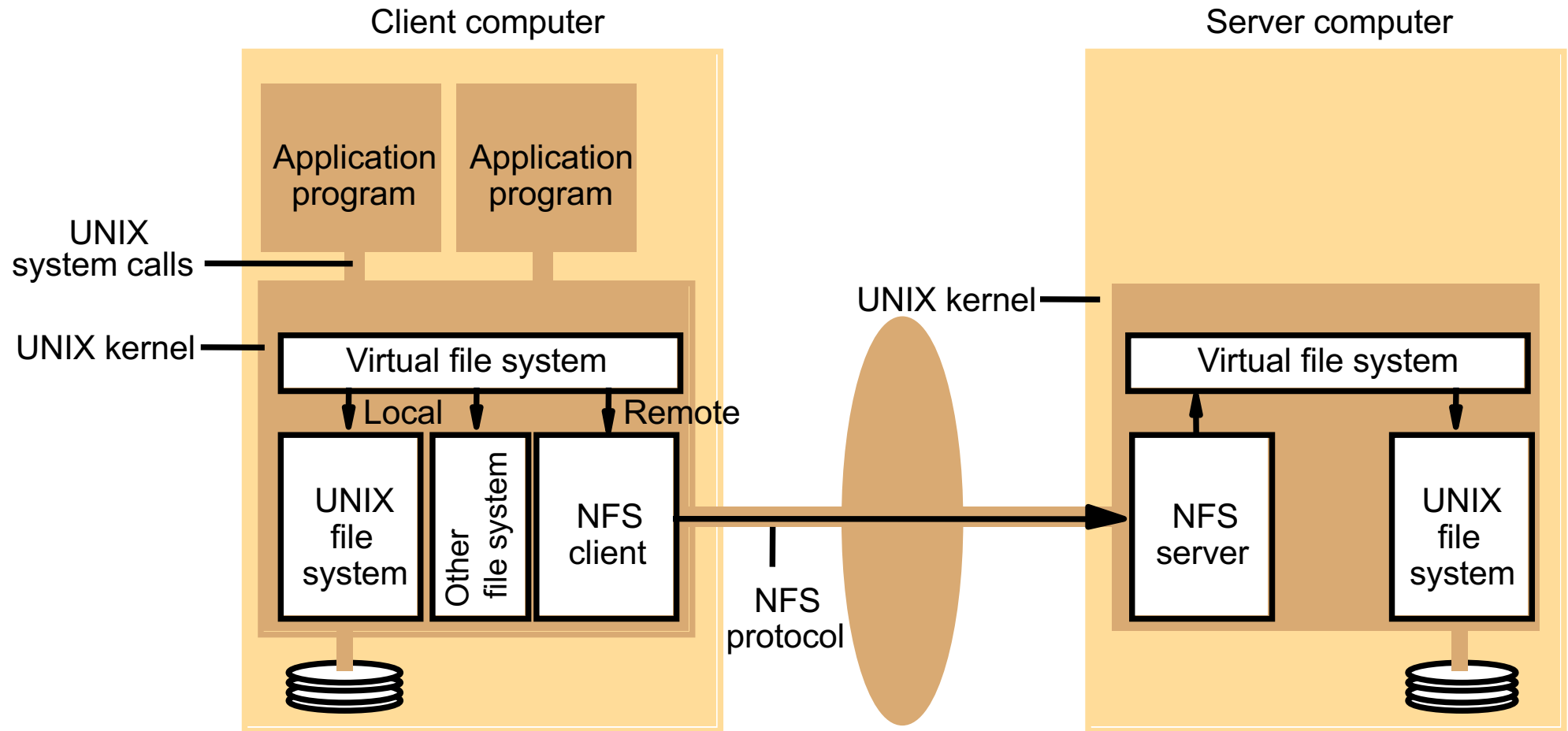
INTEGRAÇÃO NO SISTEMA LOCAL (CONT.)

Como fazer a integração ?

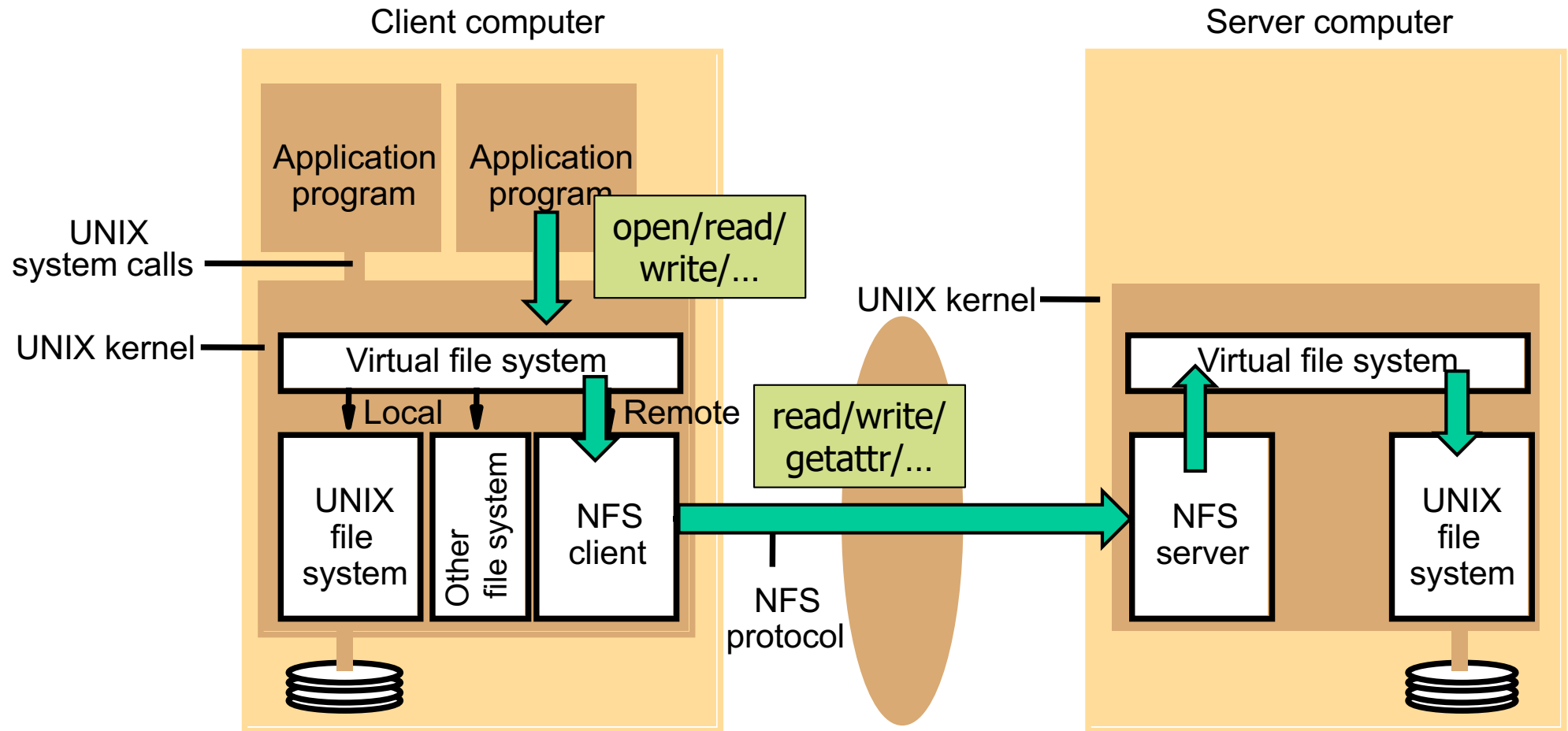
Solução: os sistemas de operação normalmente suportam mecanismos para definir novos sistemas de ficheiros

- Unix-like: VFS (Virtual File System)
- Windows: IFS (Installable Filesystem)

ARQUITECTURA



ARQUITECTURA



MODELOS DE *CACHING* : INFORMAÇÃO REPLICADA

Quase todos os sistemas de ficheiros distribuídos mantêm uma cache que pode ter diferentes características

Modelo *caching* por blocos. Guarda blocos de ficheiros (geralmente da dimensão do bloco dos discos e com dimensão idêntica à usada na transferência entre o cliente e o servidor).

Modelo *Caching* de Ficheiros “inteiros”. Guarda ficheiros completos na cache – sempre que é necessário aceder a um ficheiro, este é transferido para a máquina do cliente. Nota: *caching* efectuado normalmente em disco.

Modelo Serviço Remoto ou sem Cache. Cada vez que é necessário aceder a um ficheiro, o cliente invoca o servidor.

SUMÁRIO

Introdução à replicação

- Primário-secundário
- Multi-master

Caching

- Sistemas de ficheiros distribuídos
- Caching NFS
- Caching CIFS
- Caching Callback Promise

CACHING EM SISTEMAS UNIX

Caching de ficheiros locais em servidores UNIX

- Leituras servidas a partir da cache
- ***Read-ahead*** - carrega proativamente para cache blocos antes que sejam necessários, caso se continue a ler sequencialmente
- ***Delayed-write*** (30 seg. máximo) – escritas são feitas na cache e apenas propagadas para disco após X segundos. Porquê?
 - *Porque escrever um bloco para disco sempre que se escrevem pouco bytes seria muito ineficiente. Assim, tenta-se que quando se escreve, se escrevam todas as alterações de cada bloco.*

Aproximação semelhante noutros sistemas.

NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Cliente NFS coloca na cache blocos obtidos remotamente.

Problema: como garantir que se acede a versão atual? Porque difere da gestão de cache num servidor?

Num servidor, o SO gere todas as escritas. Logo, é impossível o valor ter mudado no disco sem o SO saber, pelo que a cache tem necessariamente a versão mais recente.

No NFS, é possível que a versão do servidor tenha sido modificada e o cliente não sabe.

NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

O servidor mantém para cada ficheiro, a seguinte informação:

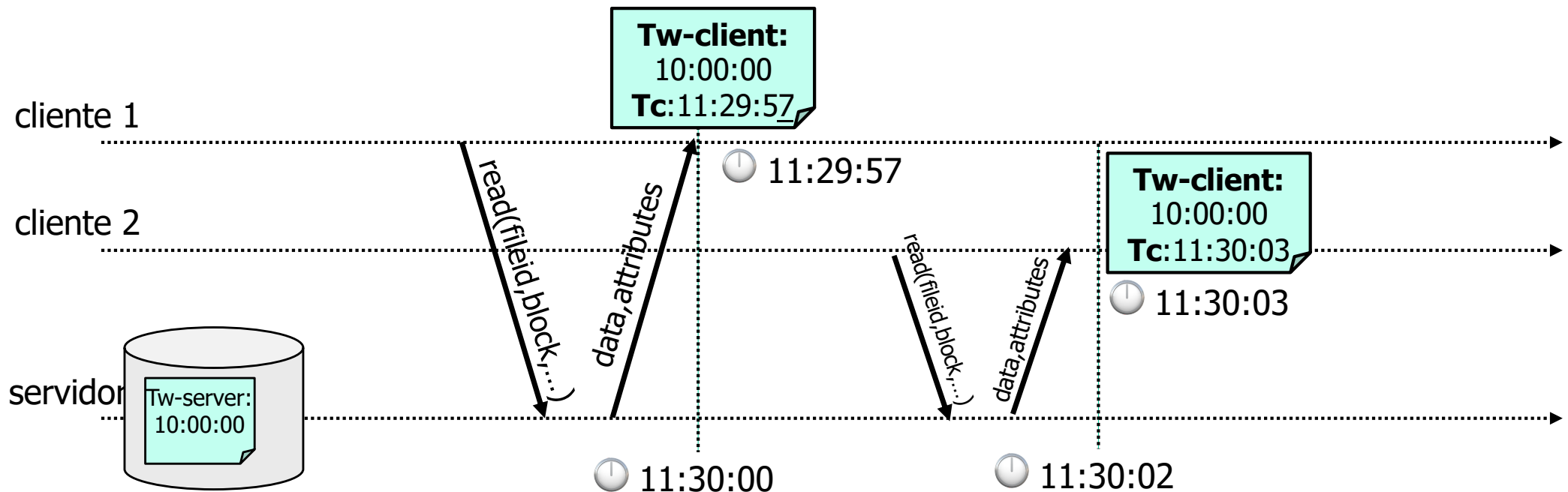
- **Tw-server** – data da última modificação



NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

O cliente mantém **para cada bloco** da cache, a seguinte informação:

- **Tc** - última hora a que a entrada na cache foi validada (hora local)
- **Tw-client** - estampilha horária da última vez que o ficheiro foi escrito (hora do servidor), guardado no cliente

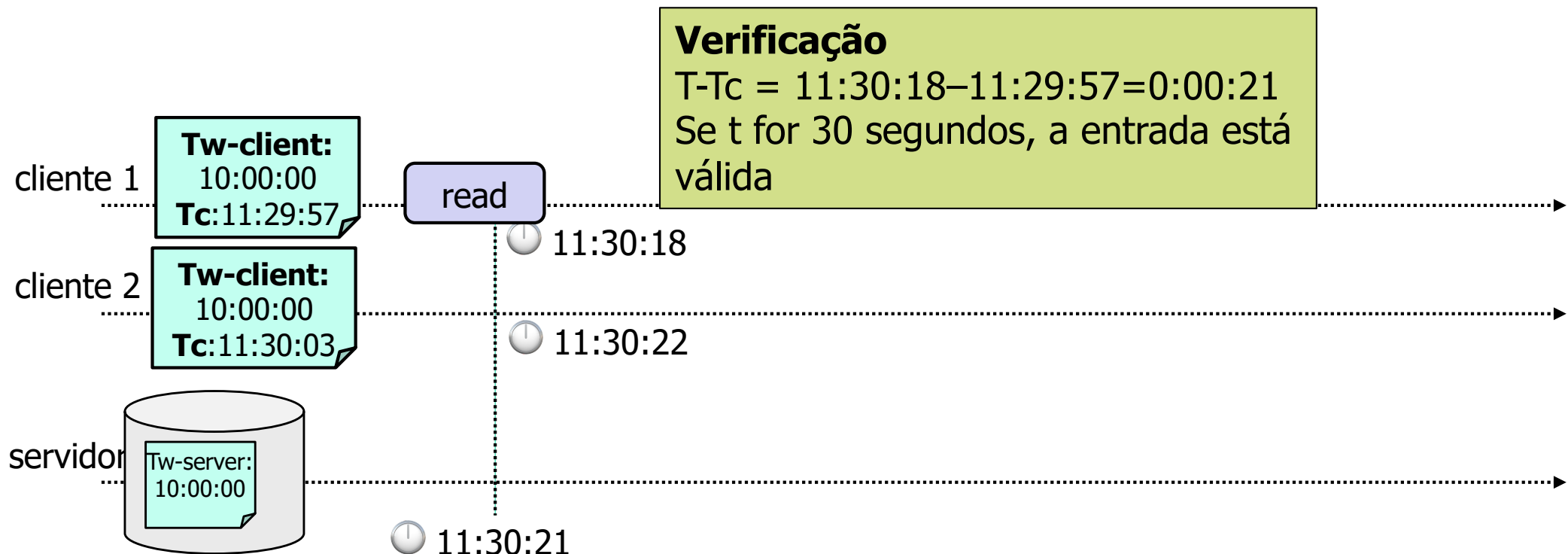


NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Quando há uma leitura, o cliente verifica se a cache se encontra válida.

No momento T , considera-se **válida** uma entrada na cache sse: $T - T_c < t$ (com t um parâmetro)

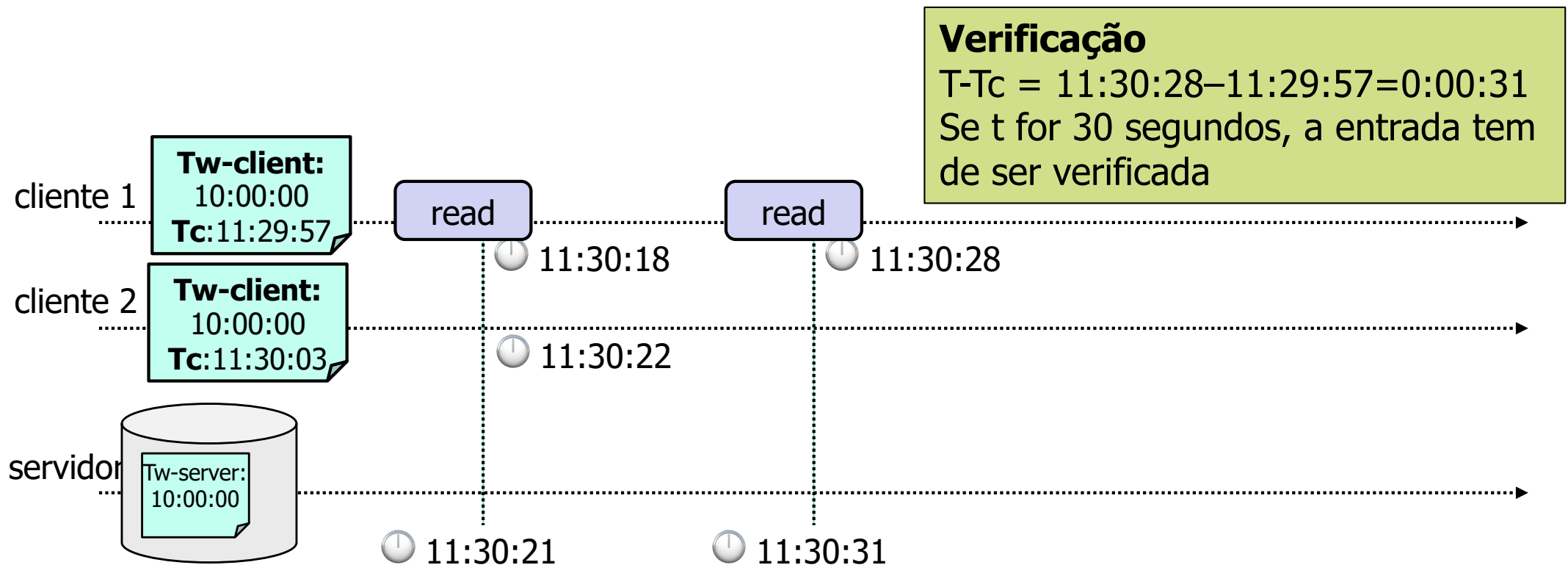
- Se t é grande, aumenta-se o risco de aceder a informação desatualizada
- Se t é pequeno, pode levar a interações com o servidor desnecessárias



NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Caso contrário, é **necessário verificar se o ficheiro não foi alterado no servidor**, i.e., se $T_{w-server} = T_{w-client}$

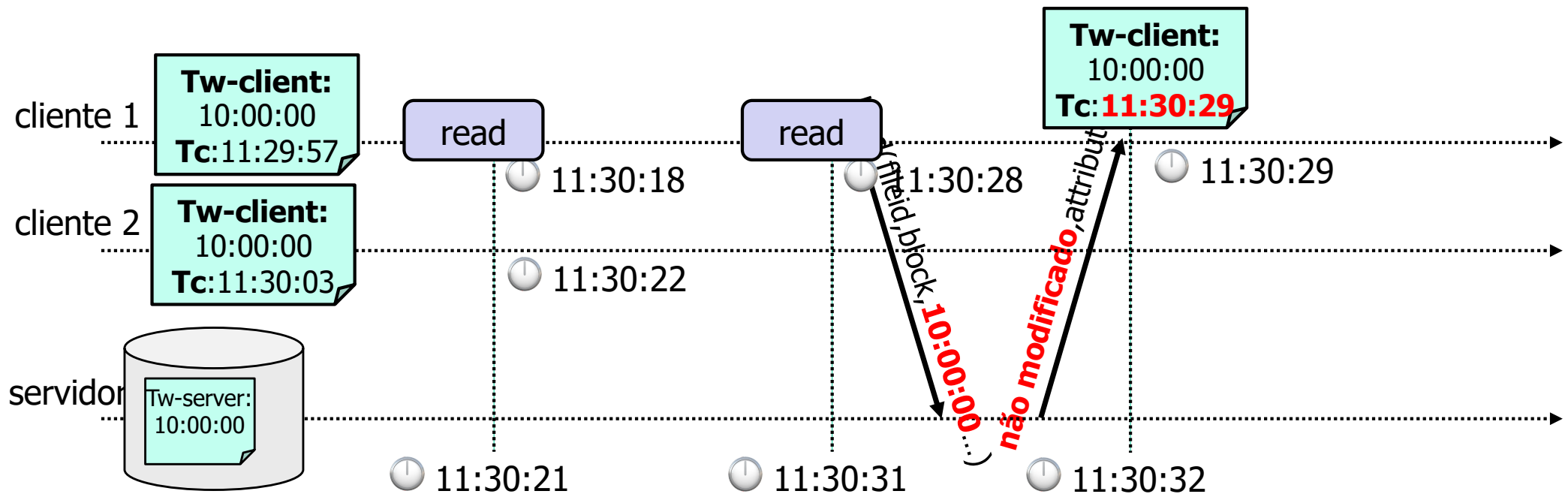
- Se $T_{w-server} \neq T_{w-client}$ todos os blocos do ficheiro são retirados da cache pois o ficheiro foi modificado.
- Caso contrário, atualiza-se o valor de T_c



NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Caso contrário, é **necessário verificar se o ficheiro não foi alterado no servidor**, i.e., se $Tw\text{-}server = Tw\text{-}client$

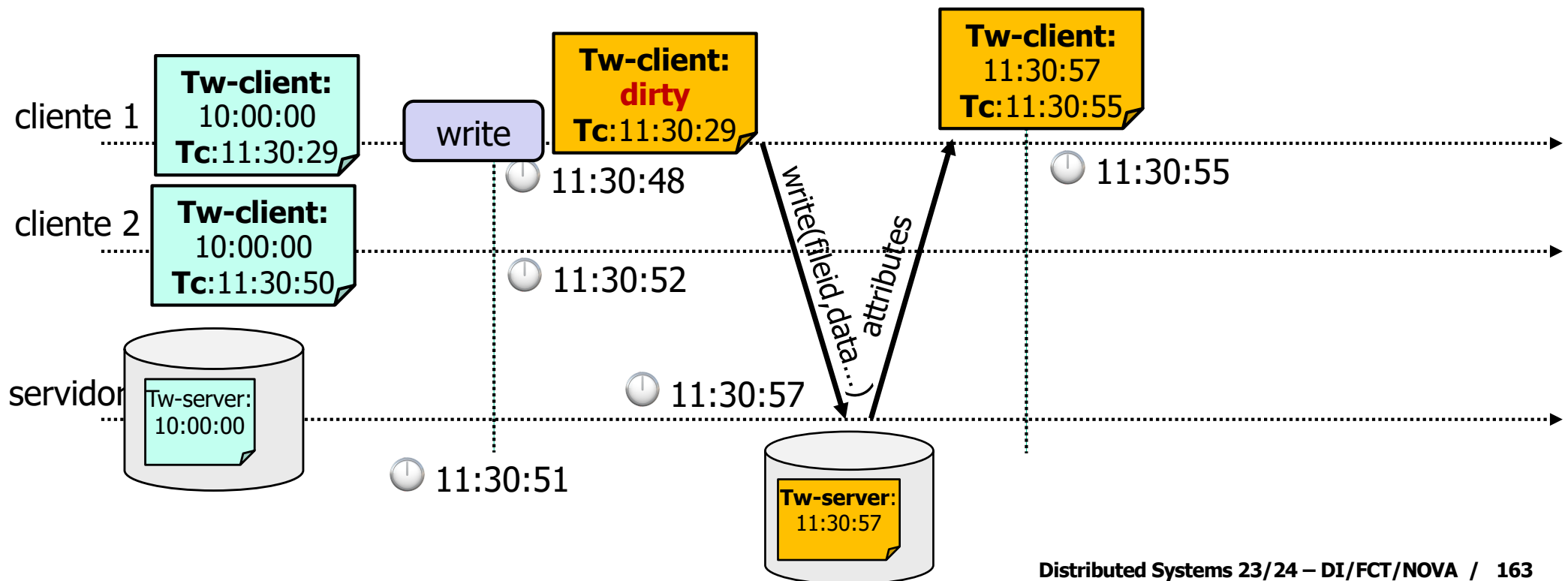
- Se $Tw\text{-}server \neq Tw\text{-}client$ todos os blocos do ficheiro são retirados da cache pois o ficheiro foi modificado.
- Caso contrário, atualiza-se o valor de Tc



NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Na escrita, os blocos modificados por uma escrita no cliente são marcados como *dirty* e enviados assincronamente para o servidor ou logo que um *sync* ocorre no cliente.

O servidor guarda os blocos com a sua data local.

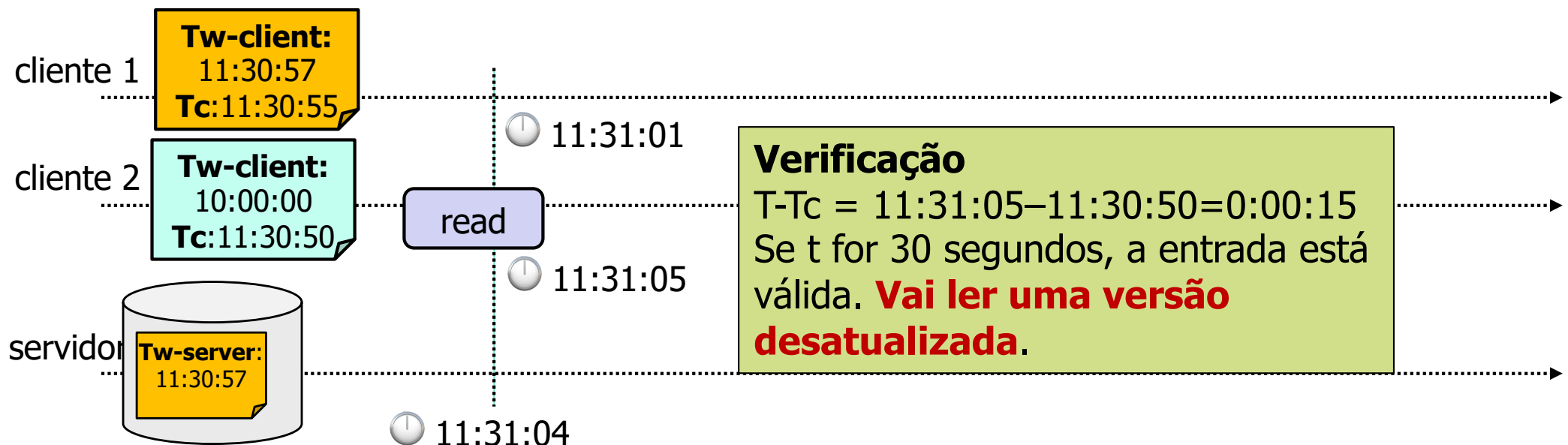


NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Na escrita, os blocos modificados por uma escrita no cliente são marcados como *dirty* e enviados assincronamente para o servidor ou logo que um *sync* ocorre no cliente.

O servidor guarda os blocos com a sua data local.

Um cliente pode ler uma versão desatualizada

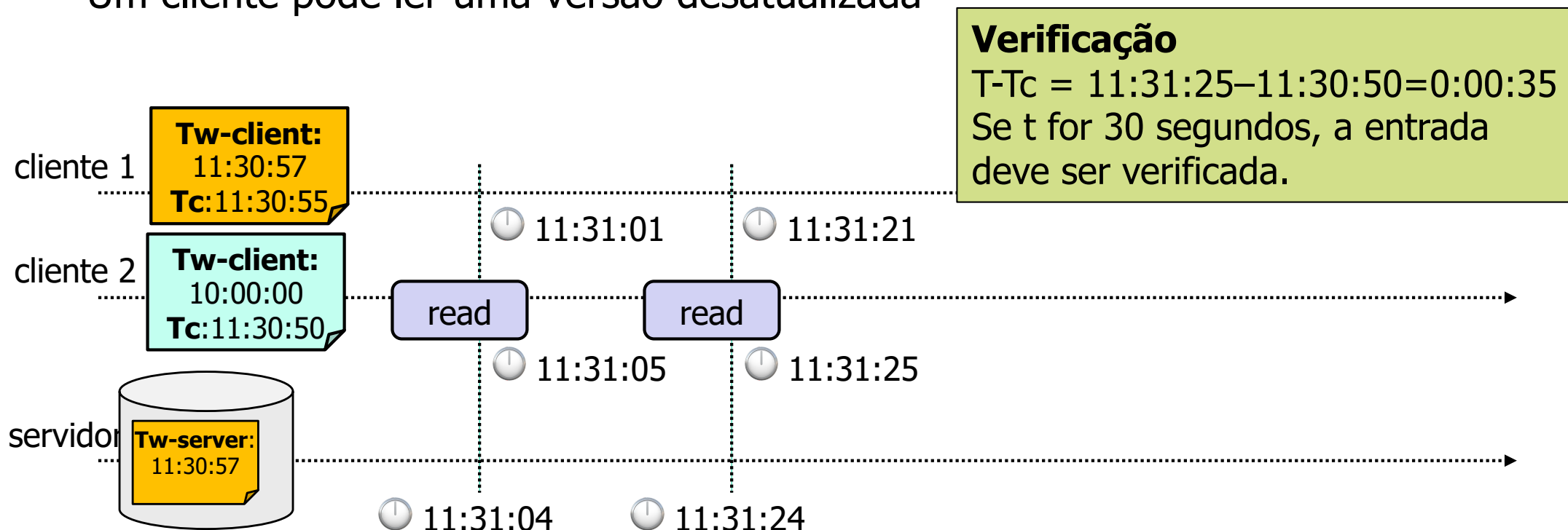


NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Na escrita, os blocos modificados por uma escrita no cliente são marcados como *dirty* e enviados assincronamente para o servidor ou logo que um *sync* ocorre no cliente.

O servidor guarda os blocos com a sua data local.

Um cliente pode ler uma versão desatualizada

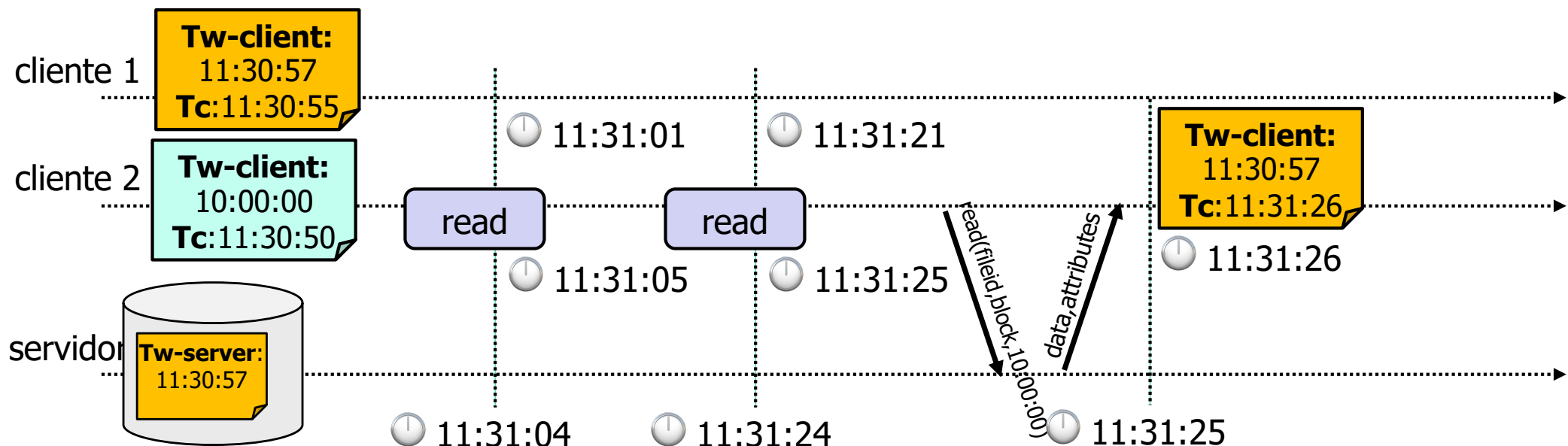


NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Na escrita, os blocos modificados por uma escrita no cliente são marcados como *dirty* e enviados assincronamente para o servidor ou logo que um *sync* ocorre no cliente.

O servidor guarda os blocos com a sua data local.

Um cliente pode ler uma versão desatualizada

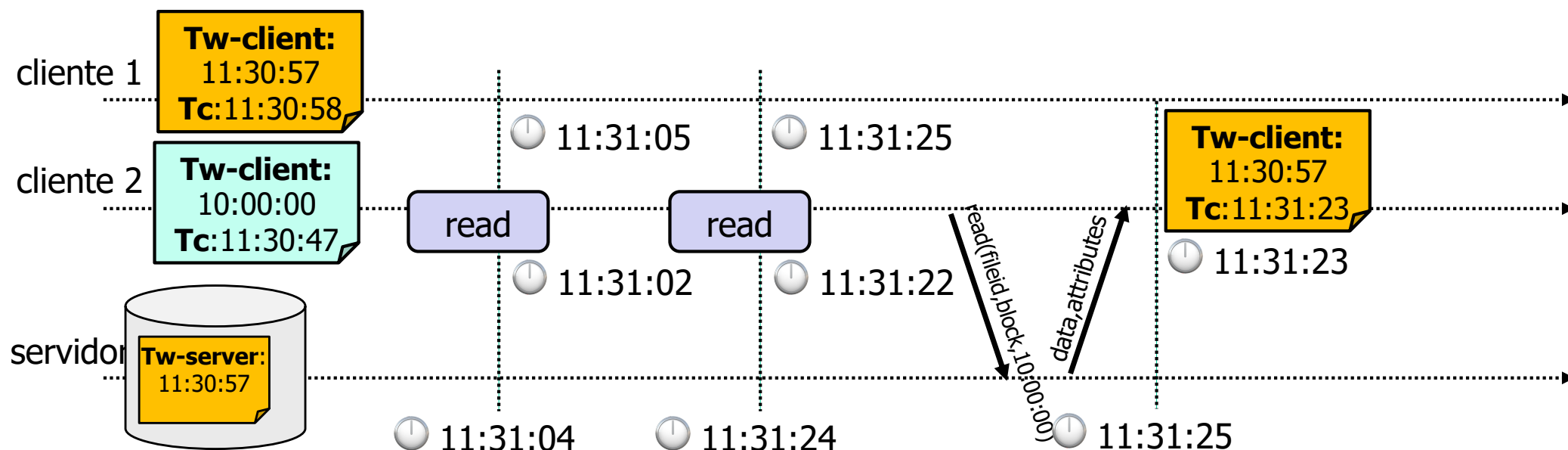


NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Na escrita, os blocos modificados por uma escrita no cliente são marcados como *dirty* e enviados assincronamente para o servidor ou logo que um *sync* ocorre no cliente.

O servidor guarda os blocos com a sua data local.

Um cliente pode ler uma versão desatualizada



NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR (3)

Para cada ficheiro, apenas existe um Tw-server

- Qual as consequências para a gestão da cache?

Como se comporta esta aproximação com um sistema de bases de dados, em que apenas alguns blocos de cada ficheiro são modificados em cada acesso?

Otimização: os atributos do ficheiro são piggybacked sempre que possível com todas as respostas do servidor, o que permite atualizar T_c

SOLUÇÕES ALTERNATIVAS: LEITURAS

Se o servidor não notifica os clientes das modificações, para garantir a frescura da informação *cached*, os clientes têm de testar a validade da cache:

- ***teste antes de cada acesso*** - aumenta a coerência da cache mas diminui o desempenho;
- **teste periódico** – pode-se estar a aceder a dados desatualizados (e.g. NFS).

SOLUÇÕES ALTERNATIVAS: ESCRITAS

A propagação de escritas pode ser efetuada em vários momentos.

Escrita "Write-through". A cada escrita no cliente corresponde uma (ou mais) escritas no servidor. Propriedades?

Favorece a fiabilidade à custa do desempenho.

Escrita "Delayed-Write". As escritas são inicialmente efetuadas na cache local e, posteriormente, enviadas para o servidor. Propriedades?

Esta aproximação diminui o tráfego de rede mas diminui a (1) fiabilidade, porque se o cliente falhar antes de se propagarem as modificações, estas perdem-se; e (2) consistência, porque outro cliente que vá ao servidor não tem acesso às escritas que estão na cache do cliente.

SUMÁRIO

Introdução à replicação

- Primário-secundário
- Multi-master

Caching

- Sistemas de ficheiros distribuídos
- Caching NFS
- Caching CIFS
- Caching Callback Promise

GESTÃO DE CACHE NO SISTEMA CIFS: OPPORTUNISTIC LOCKS

Neste esquema, introduzido no sistema CIFS (sucessor do SMB), existem vários tipos de *locks*:

Opportunistic locks (oplocks) exclusivos que permitem ao cliente ter acesso exclusivo ao ficheiro e fazer *caching* arbitrário do mesmo. Um *oplock* pode ser retirado ao cliente pelo servidor.

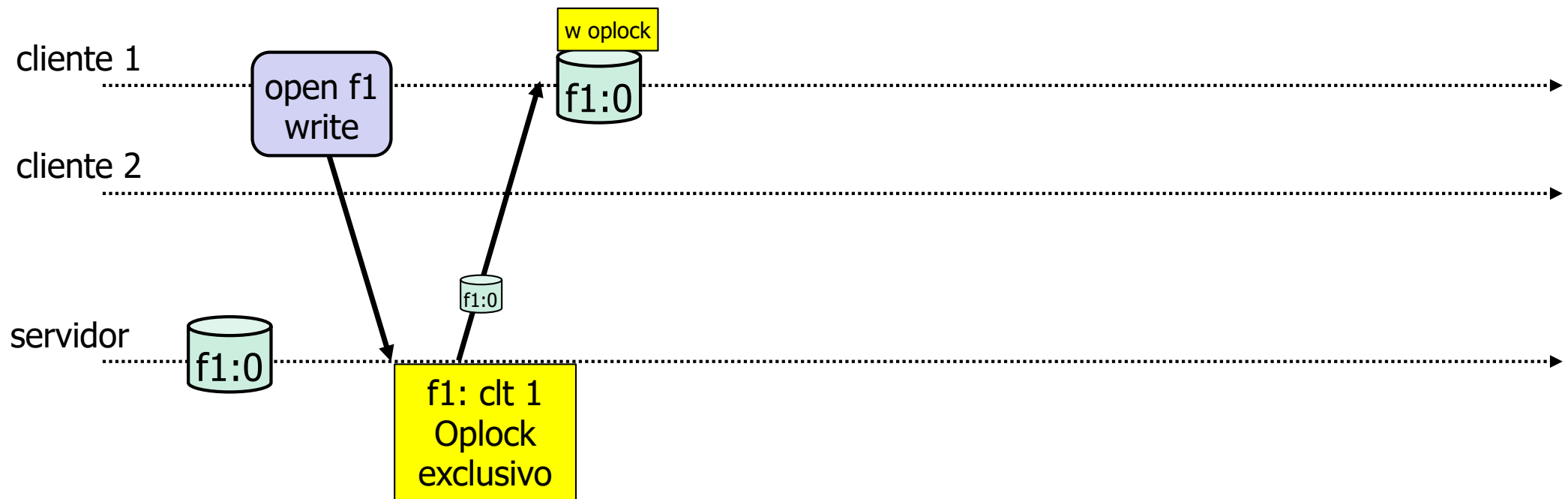
Oplocks partilhados que permitem aos clientes ter acesso ao ficheiro em leitura e fazer *caching* arbitrário do mesmo. Um *oplock* pode ser retirado ao cliente pelo servidor.

Mandatory locks que permitem acessos exclusivos e *caching* e que não podem ser retirados pelo servidor

Os clientes fazem ou não *caching* dos ficheiros conforme o tipo de *lock* que têm. Se não têm nenhum podem sempre aceder ao ficheiro mas sem fazer *caching* do mesmo.

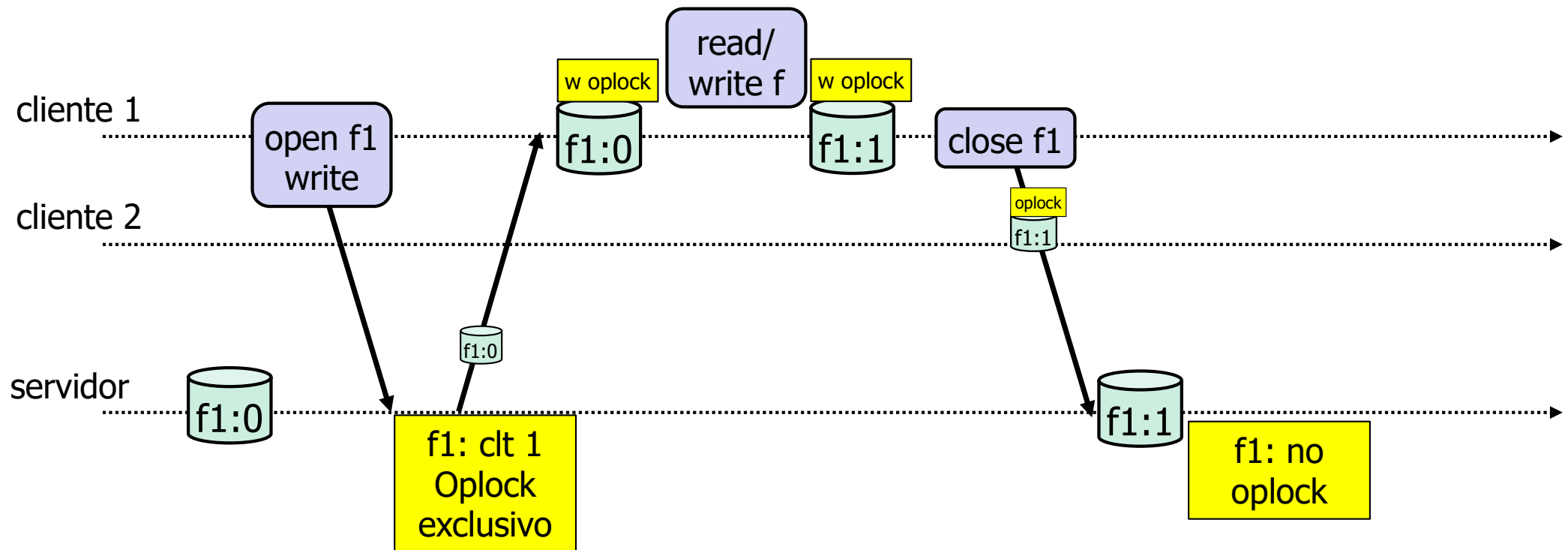
GESTÃO DOS OPLOCKS – CENÁRIO 1

1. Ao aceder a um ficheiro, cliente pede:
 1. **Oplock exclusivo** caso pretenda **escrever** o ficheiro
 2. **Oplock partilhado** caso pretenda apenas **ler** o ficheiro
2. Caso nenhum outro cliente tenha um **oplock** sobre o ficheiro, servidor concede o **oplock** pedido e envia o ficheiro, que pode ser cached pelo cliente.



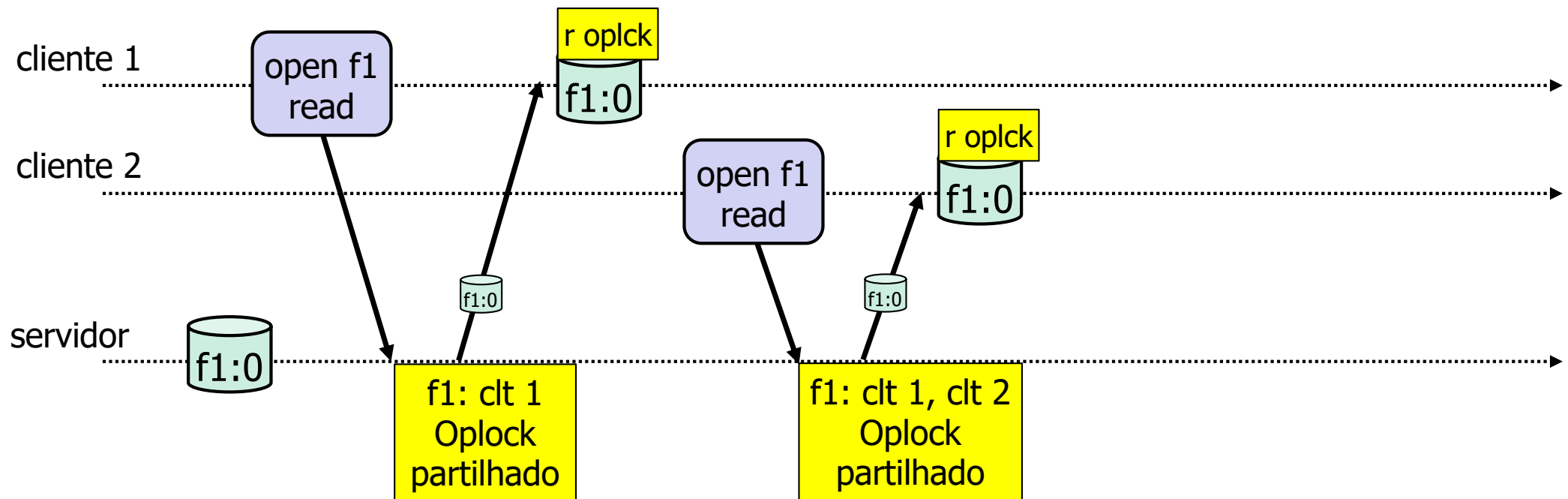
GESTÃO DOS OPLOCKS – CENÁRIO 1

1. Operações de leitura ou escrita (com oplock exclusivo) podem ser locais.
2. Quando fecha ficheiro, propaga novo estado e liberta oplock



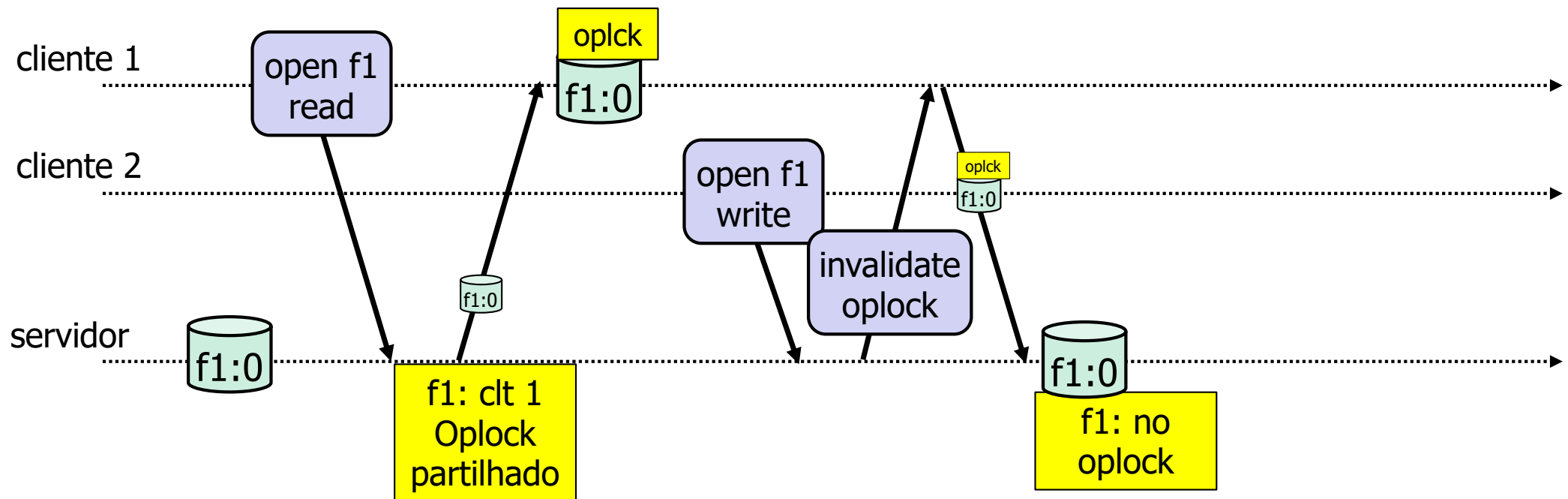
GESTÃO DOS OPLOCKS – CENÁRIO 2

1. Cliente 1 abre ficheiro para leitura – pede **oplock partilhado**.
2. Cliente 2 abre ficheiro para leitura – pede **oplock partilhado**.
3. Servidor concede **oplock partilhado** ao cliente 2.
4. Leituras podem ser locais.



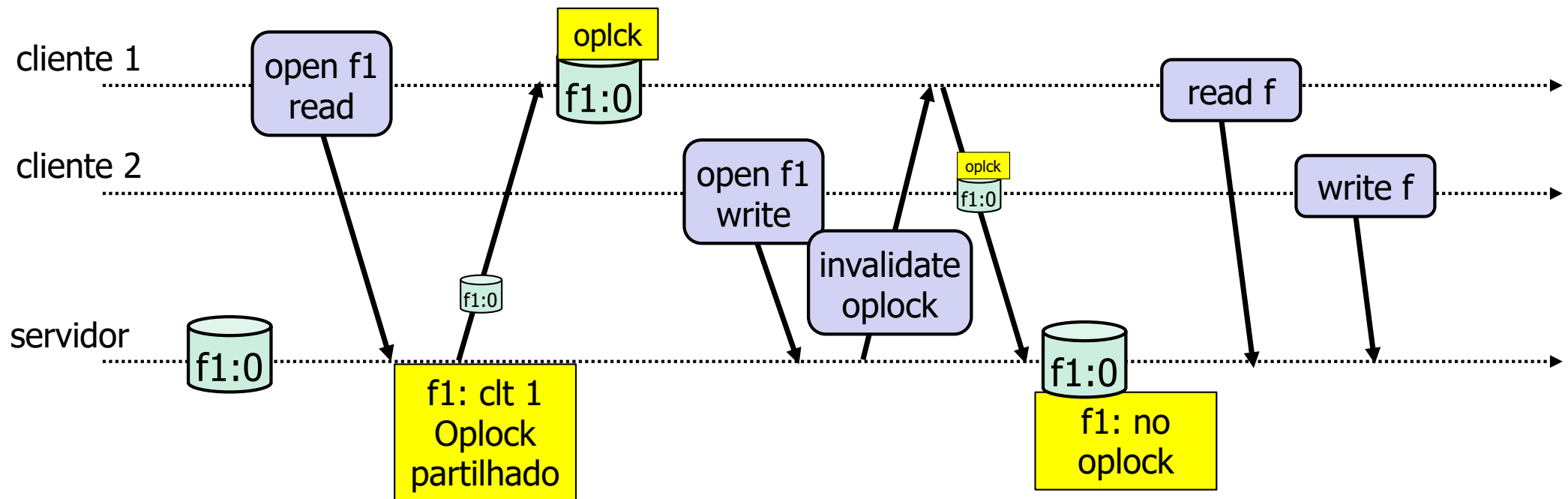
GESTÃO DOS OPLOCKS – CENÁRIO 3

1. Cliente 1 abre ficheiro para leitura – pede **oplock partilhado**.
2. Cliente 2 abre ficheiro para escrita – pede **oplock exclusivo**.
3. Servidor invalida oplock do cliente 1 e nenhum cliente fica com oplock. Caso cliente 1 tenha oplock exclusivo, deve enviar alterações efetuadas.



GESTÃO DOS OPLOCKS – CENÁRIO 3

Leituras e escritas têm de ir ao servidor.



GESTÃO DE OPLOCKS : RESUMO

- **N clientes de leituras** => todos os clientes têm um *oplock partilhado* e podem fazer cache do ficheiro (sem necessidade de verificar se foi atualizado no servidor)
- **1 cliente de escrita** => o clientes tem um *oplock exclusivo* e pode fazer cache do ficheiro (sem necessidade de verificar se foi atualizado no servidor)
- **Mais do que um cliente, sendo 1 cliente de escrita** => nenhum cliente tem um *oplock* e portanto nenhum cliente pode fazer cache – acesso feito contactando o servidor.

GESTÃO DOS OPLOCKS (CONT.)

Quando um ficheiro é aberto para leitura/escrita tenta-se prolongar concorrência, assumindo que o cliente é de leitura até à primeira escrita – nesse momento, efetua-se o procedimento de adição de um novo cliente.

Se um *oplock* é quebrado, deixa de haver *caching* do ficheiro.

SUMÁRIO

Introdução à replicação

- Primário-secundário
- Multi-master

Caching

- Sistemas de ficheiros distribuídos
- Caching NFS
- Caching CIFS
- Caching Callback Promise

SOLUÇÃO “CALLBACK PROMISE”

- Introduzido no AFS e usado também no Coda.

Granularidade de replicação

Neste sistema os clientes fazem *caching* de ficheiros inteiros, sendo o ficheiro a unidade de transferência entre o cliente e o servidor.

A cache é mantida em disco – o objetivo é garantir que a cache se mantém quando o cliente se desliga e volta a ligar.

Manutenção da coerência da cache

Quando um cliente obtém uma cópia do ficheiro para fazer cache, o servidor *promete* informar o cliente de qualquer modificação efetuada ao ficheiro – a esta promessa chama-se uma “***callback promise***”.

Desde que o cliente tenha uma “***callback promise***” **válida**, assume que a sua cópia do ficheiro está atualizada. Neste caso, um ficheiro é aberto no cliente sem nenhuma comunicação com o servidor.

SOLUÇÃO "CALLBACK PROMISE"

Semântica de acesso – semântica de sessão

Um cliente acede à versão do ficheiro que existe no cliente quando o ficheiro é aberto.

Leituras e escritas acedem a essa versão.

Quando o ficheiro é fechado no cliente, o cliente enviará assincronamente a nova versão do ficheiro para o servidor.

Esta aproximação garante que o cliente acede a uma versão do ficheiro que não será alterada durante o período em que se está a aceder ao ficheiro.

SOLUÇÃO “CALLBACK PROMISE”

Integração de escritas no servidor

Os cliente propaga novas versões dos ficheiros para os servidores.

Os servidores verificam se houve alterações concorrentes (usando vetores-versão). Caso haja uma alteração concorrente, há duas alternativas:

- Marca o ficheiro como em conflito;
- Executa um programa para unificar as alterações concorrentes.

O **servidor notifica todos os clientes** com “callback promises” válidas.

Um cliente, ao receber a notificação, anula a sua “callback promise” e, se o ficheiro estiver aberto, obtém uma nova cópia do ficheiro.

Integração de escritas no servidor

Após ter estado desligado/desconectado, um cliente tem de revalidar as “*callback promises*” que tem. Porquê?

Pode ter perdido os “callbacks” feitas pelo servidor enquanto o cliente esteve desligado.

PARA SABER MAIS

G. Coulouris, J. Dollimore and T. Kindberg, Blair Gordon,
Distributed Systems - Concepts and Design, Addison-Wesley,
5th Edition, 2011

- Capítulo 12.1-12.4 e 18.3-18.4
- CIFS/SMF: slides

Giuseppe DeCandia, et. al. 2007. Dynamo: amazon's highly
available key-value store. In SOSP'07, 2007.

<https://doi.org/10.1145/1294261.1294281>