

# SISTEMAS DISTRIBUÍDOS

## Capítulo 6

### 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

# MOTIVAÇÃO PARA UTILIZAÇÃO DE SISTEMAS DE FICHEIROS DISTRIBUÍDOS

Acesso a dados a partir de diferentes dispositivos

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 semelhante ou superior

# 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.

# ALGUMAS QUESTÕES ENVOLVIDAS NA DISTRIBUIÇÃO

**Integração:** como integrar no sistema de ficheiro local?

**Designação:** como designar os ficheiros remotos?

Externamente: ao nível das aplicações

Internamente: no sistema

**Modelo de acesso:** dois extremos: acesso completamente remoto ou acesso completamente local (usando *cache*).

**Partilha de ficheiros:** problemas relacionados com o controlo da concorrência e modelo de consistência fornecido

# INTEGRAÇÃO NO SISTEMA LOCAL

Os ficheiros dum sistema distribuído de ficheiros devem ser acedidos localmente numa máquina

Problema: Qual a granularidade da partilha?

Solução típica: sub-árvore do sistema de ficheiros

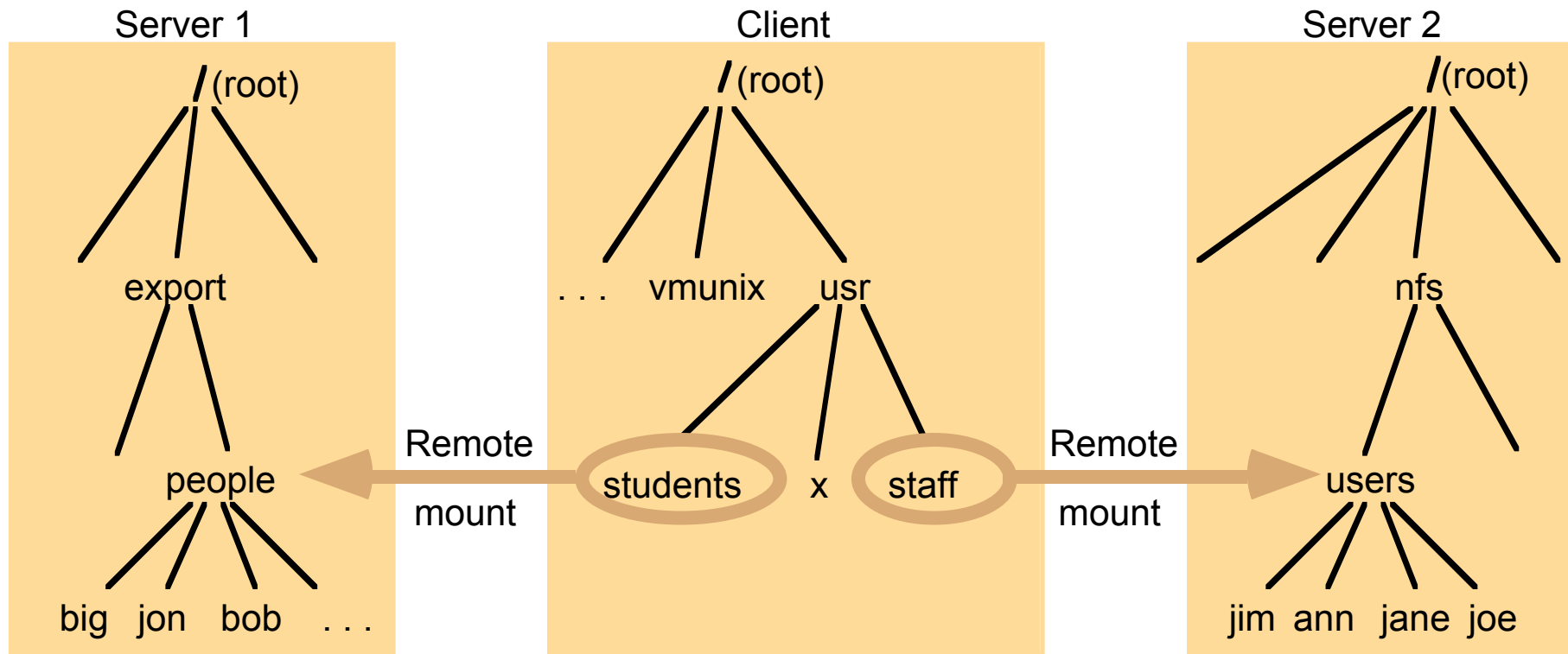
Problema: Como providenciar acesso aos ficheiros?

Solução: integrar a sub-árvore na hierarquia local

Windows: definindo uma nova *drive*

Unix-like: através do mecanismo de *mount*

# EXEMPLO: MOUNT REMOTO E DESIGNAÇÃO NO NFS



Cada servidor exporta um conjunto de directorias.

O cliente pode montar uma directoria remota (caso tenha permissões) numa directoria local (ex.: `/usr/students` no cliente monta a árvore `/export/people` no Server 1)

# INTEGRAÇÃO NO SISTEMA LOCAL (CONT.)

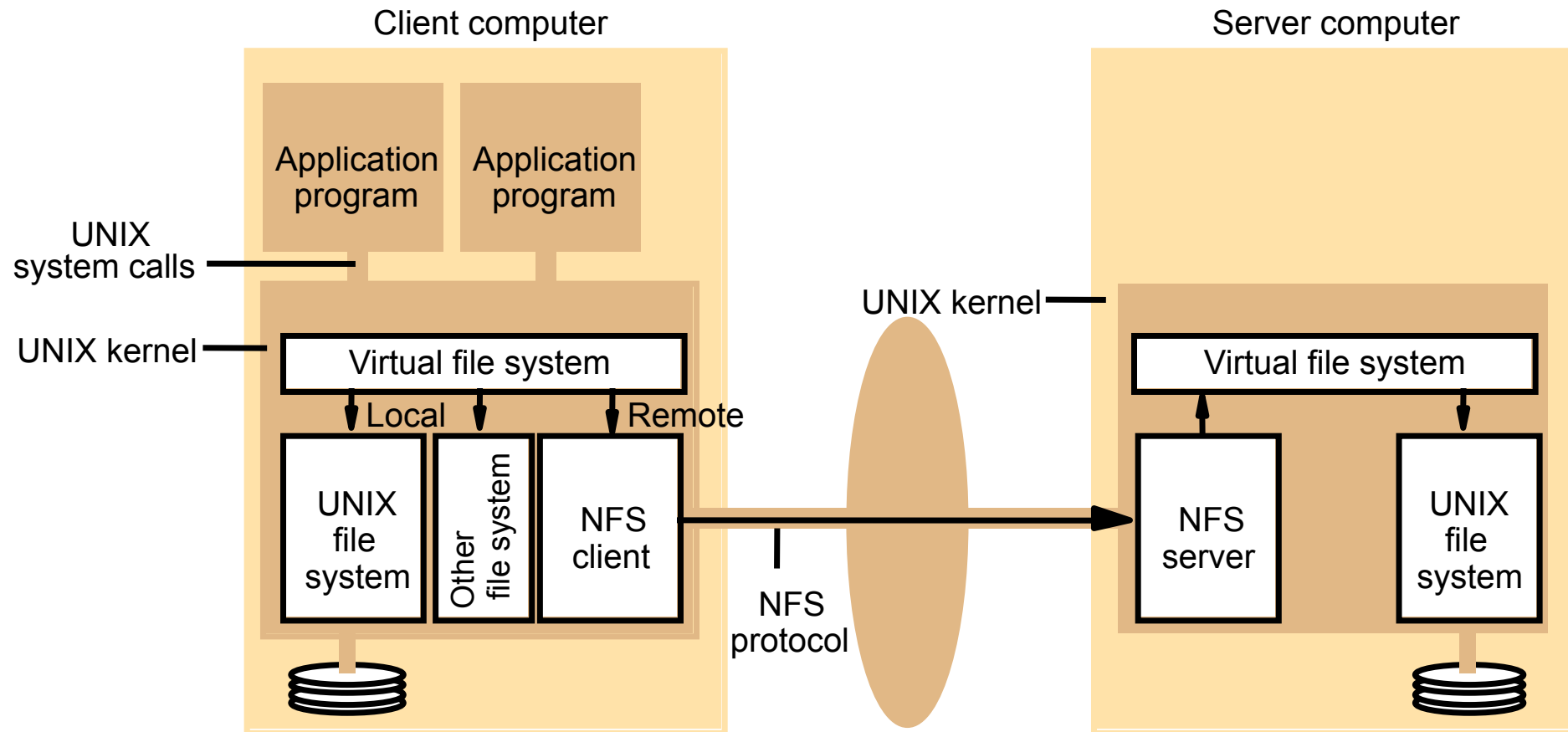
Como fazer a integração ?

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

Unix-like: VFS

Windows: IFS

# ARQUITECTURA



# DESIGNAÇÃO DOS FICHEIROS

Tipos dos nomes dos ficheiros:

Nomes simbólicos

(exemplo: /x/y/p/q )

Nomes internos ou sistema

(exemplos: FileId, File-handles, UFIDs)

# TIPOS DOS NOMES SIMBÓLICOS

**Nomes globais.** Podem ser passados entre máquinas mas implicam a introdução de uma super-raiz do *file system*.

Uma forma popular é: “//servername/ficheiro”

Exemplo: //asc.di.fct.unl.pt/home/foobar

**Nomes contextuais.** Todos os nomes são interpretados a partir de um *file system* local.

Fáceis de implementar mas mais difícil passar um nome entre duas máquinas

Ligações (mounts)

Um mesmo nome em duas máquinas pode referir o mesmo ou diferentes ficheiros

Nomes diferentes em duas máquinas podem referir o mesmo ficheiro

# CARACTERÍSTICAS DOS NOMES INTERNOS

Para que os clientes e os servidores possam memorizar UFIDs (ou *file handles*) sem que os mesmos possam dar origem a ambiguidades, estes devem ser:

- únicos no tempo

- únicos no espaço

Que propriedades tem um NFS *file handle* ?

[ file system number, file i-node, i-node generation number ],  
endereço IP do servidor, porta do servidor

# MODELOS DE *CACHING* : INFORMAÇÃO REPLICADA

Quase todos os sistemas de ficheiros distribuídos mantêm um 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.

## **Eficiência?**

Depende da fracção de operações que podem ser servidas pelos valores guardados na cache.

Caching fich. inteiros > caching blocos > sem caching

# MODELOS DE *CACHING* : INFORMAÇÃO REPLICADA

Quase todos os sistemas de ficheiros distribuídos mantêm um 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.

## **Coerência?**

Depende do modo como é verificada a validade da cache  
Sem caching > caching blocos e caching fich. inteiros

# SEMÂNTICA DO ACESSO NUM SISTEMA DE FICHEIROS DISTRIBUÍDOS

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 esquema 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 servidor.

# NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

Os servidores Unix e os respectivos clientes usam sempre *caching* de ficheiros locais na cache, em memória central. A política convencional é satisfazer os pedidos de leitura da cache se possível e usar *read-ahead* e *delayed-write* (30 segundos no máximo) dos ficheiros locais.

O módulo cliente do NFS coloca também na cache local os blocos que lê do servidor e utiliza um sistema de *timestamps* para controlar a respectiva validade. Há duas *timestamps* associadas a cada bloco na cache:

- Tc - última hora a que esta entrada na cache foi validada (hora local)

- Tw-client - estampilha horária da última vez que o ficheiro foi escrito (hora do servidor tal como está registada actualmente no cliente)

No momento T, considera-se válida uma entrada na cache sse:  $T - T_c < t$  Caso contrário, é necessário verificar se o ficheiro não foi alterado no servidor, i.e., se  $T_w\text{-server} = T_w\text{-client}$

Se t é grande pode resultar incoerência, se t é pequeno pode resultar ineficiência.

# NFS: GESTÃO DE CACHE SEM ESTADO NO SERVIDOR (2)

O valor de  $t$  é calculado de forma dinâmica e varia entre 3 e 30 segundos para ficheiros (o valor é menor quanto mais frequentemente o ficheiro é escrito) e entre 30 e 60 segundos para directórios (estratégia idêntica).

Se  $T - T_c < t$ , o segundo teste não é realizado, caso contrário é executado um *getattr request* ao servidor e os valores de  $T_c$  e  $T_w$  são actualizados.

Se  $T_{w-server} \neq T_{w-client}$  todos os blocos do ficheiro são retirados da cache pois o ficheiro foi modificado.

Para otimizar, os valores de  $T_w$  são aplicados a todos os blocos do mesmo ficheiro e os atributos do ficheiro são piggybacked sempre que possível com todas as respostas do servidor (ver a definição das operações NFS)

Qual o problema de manter apenas um  $T_w$  para cada ficheiro?

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.

Como se comporta esta aproximação com um sistema de bases de dados?

# GESTÃO DE CACHE SEM ESTADO NO SERVIDOR

**Leituras.** Para garantir a frescura da informação *cached*, os clientes de servidores *stateless* têm de testar a validade da mesma:

teste antes de cada acesso - aumenta a coerência mas diminui a performance  
teste periódico - semântica sem garantias totais

**Escrita.** A propagação de escritas pode ser efectuada em vários momentos.

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

Favorece a fiabilidade e a semelhança com a semântica tradicional tipo "sistema equivalente a uma só cópia" à custa do sacrifício do desempenho.

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

Estes esquemas diminuem o tráfego de rede mas diminuem a fiabilidade e consistência. Quando o servidor é *stateless* e ignora os clientes, não é possível implementar a semântica "equivalência a uma só cópia".

# 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.

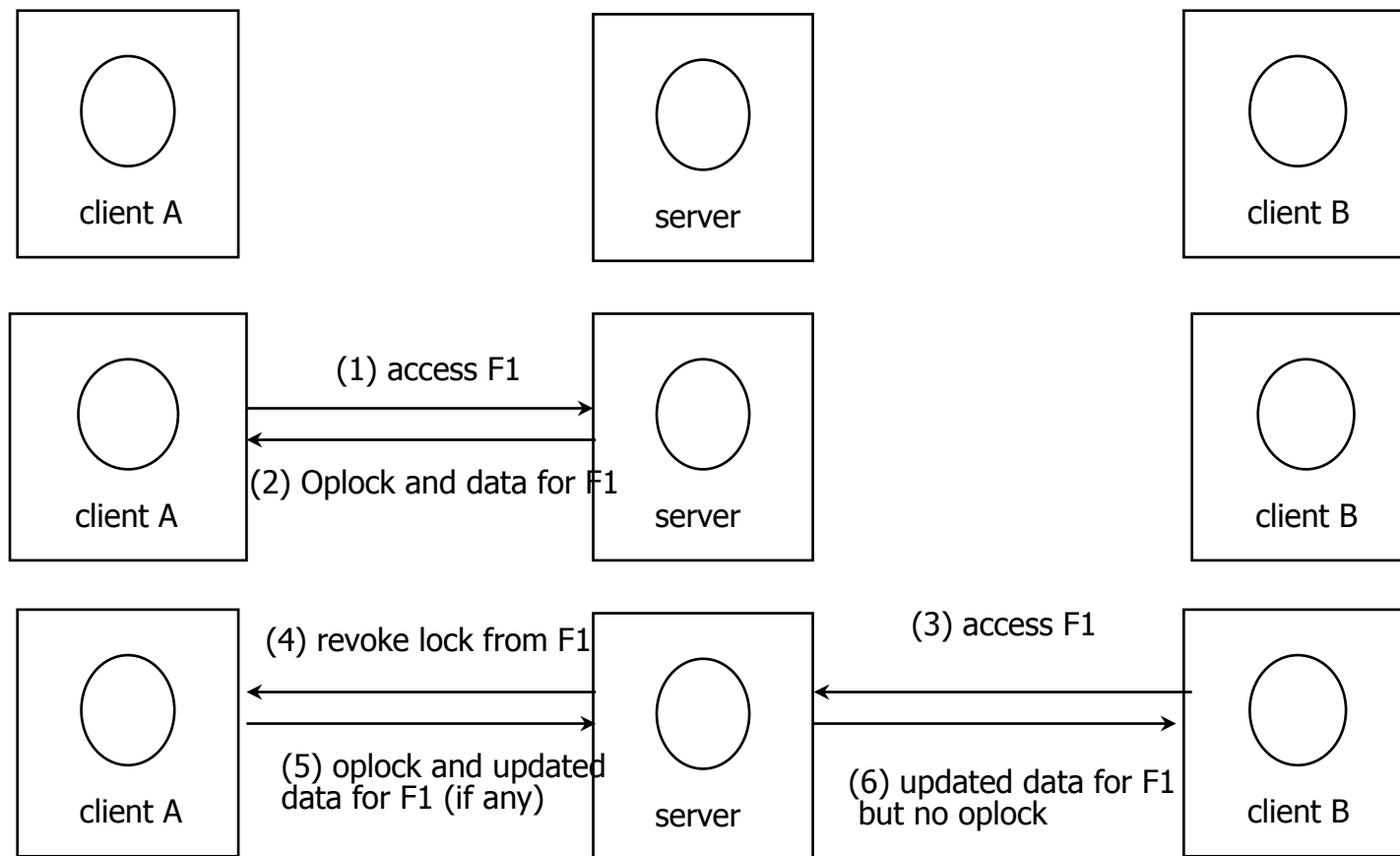
*Oplocks partilhados* que permitem aos clientes ter acesso ao ficheiro em leitura e fazer *caching* arbitrário do mesmo. Um *oplock* pode ser retirado ao cliente pelo servidor.

*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.

# ILUSTRAÇÃO

Neste exemplo ilustra-se a utilização de *oplocks* para acesso ao ficheiro F1 por dois clientes A e B.



# GESTÃO DOS OPLOCKS

1. O primeiro cliente obtém:
  1. Oplock exclusivo caso pretenda escrever o ficheiro
  2. Oplock partilhado caso pretenda apenas ler o ficheiro
2. Quando surgem novos clientes de leitura:
  1. Existe um cliente com oplock exclusivo  
Cliente perde lock e deve enviar as modificações efectuadas  
Ambos os clientes ficam sem oplocks (e portanto sem poder fazer cache)
  2. Existe um cliente com oplock partilhado  
Ambos os clientes ficam com oplock partilhado
3. Quando surge um novo cliente de escrita
  1. Existe um cliente com oplock exclusivo  
Cliente perde lock e deve enviar as modificações efectuadas  
Ambos os clientes ficam sem oplocks (e portanto sem poder fazer cache)
  2. Existe um (ou mais) clientes com oplock partilhado  
Clientes perdem o oplock  
Todos os clientes ficam sem oplocks (e portanto sem poder fazer cache)

# 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, efectua-se o procedimento de adição de um novo cliente.

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

# SOLUÇÃO “CALLBACK PROMISE”

Introduzido no AFS e usado também no Coda.

Neste sistema os clientes fazem *caching* de ficheiros inteiros, sendo o ficheiro a unidade de transferência entre o cliente e o servidor. Quando um cliente obtém um cópia do ficheiro, o servidor *promete* informar o cliente de qualquer modificação efectuada 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 actualizada. Neste caso, um ficheiro é aberto no cliente sem nenhuma comunicação com o servidor.

Quando um programa no cliente fecha um ficheiro que acabou de modificar, o cliente AFS/Coda envia as modificações para o servidor. Nessa altura, 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.

# SOLUÇÃO “CALLBACK PROMISE”

Com este esquema todas as escritas feitas no cliente são temporárias até ao fecho do ficheiro. Desta forma, a única semântica que pode ser implementada é uma semântica dita “semântica de sessão”. Com esta semântica, uma leitura lê o resultado das escritas feitas depois do último *close* do ficheiro.

Quando um cliente é inicializado, ele tem de revalidar as “*callback promises*” que tem. Porquê?

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

# INTRODUÇÃO À REPLICAÇÃO: MOTIVAÇÃO

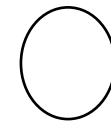
Tolerar falhas

Distribuir carga

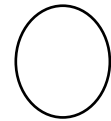
# PRIMÁRIO / SECUNDÁRIO (PASSIVO)

Primário: mantém versão oficial dos dados

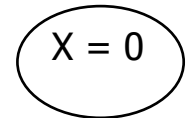
Secundário: mantém cópia da versão do primário



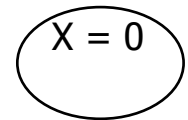
cliente



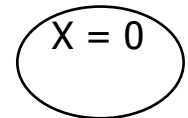
cliente



secundário



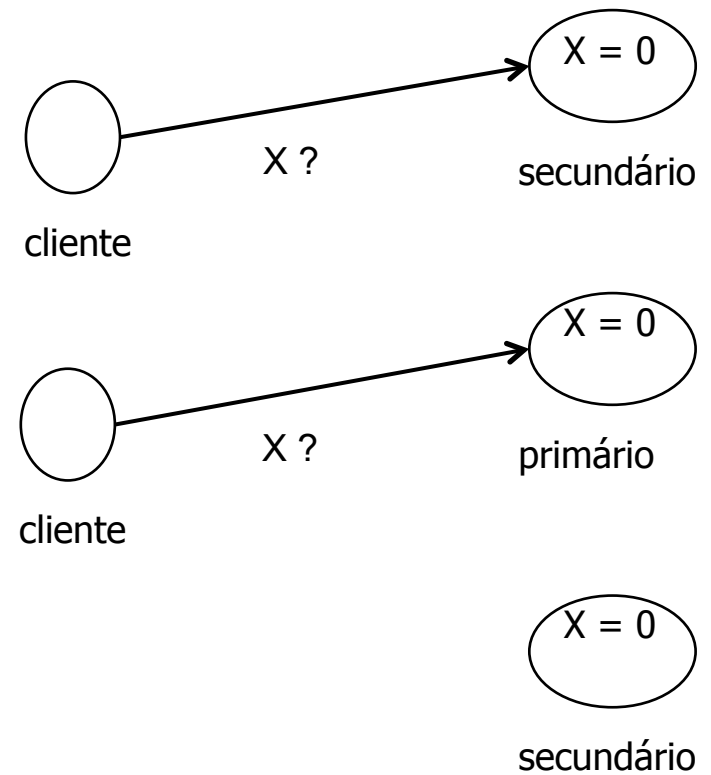
primário



secundário

# PRIMÁRIO / SECUNDÁRIO : LEITURA

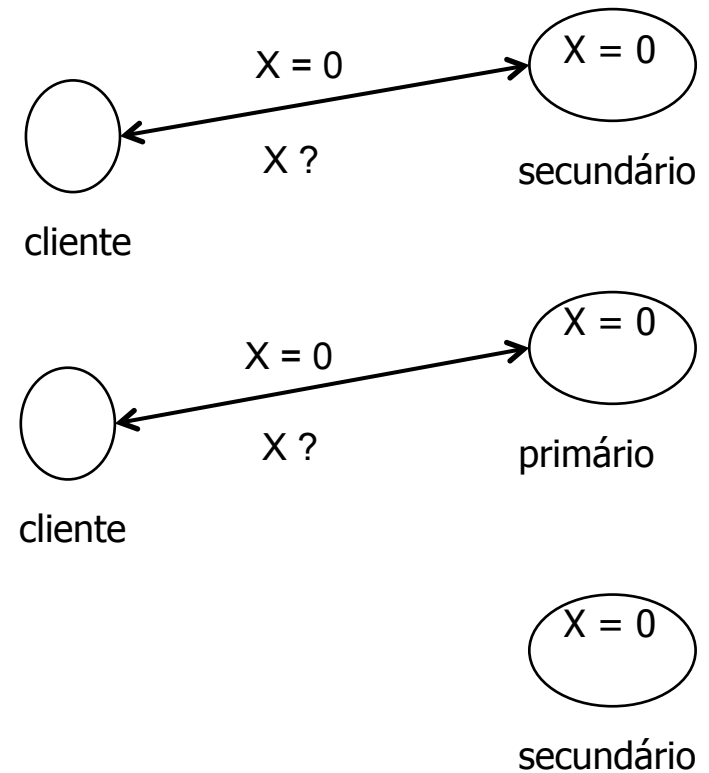
Leitura enviada para qualquer réplicas



# PRIMÁRIO / SECUNDÁRIO : LEITURA

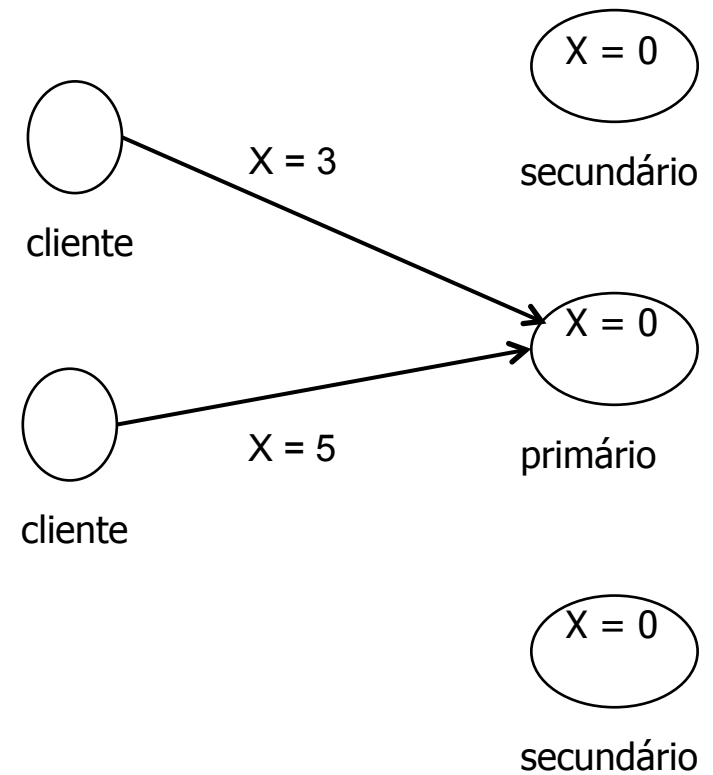
Leitura enviada para qualquer réplicas

Secundário executa operação e devolve resultado



# PRIMÁRIO / SECUNDÁRIO : ESCRITA

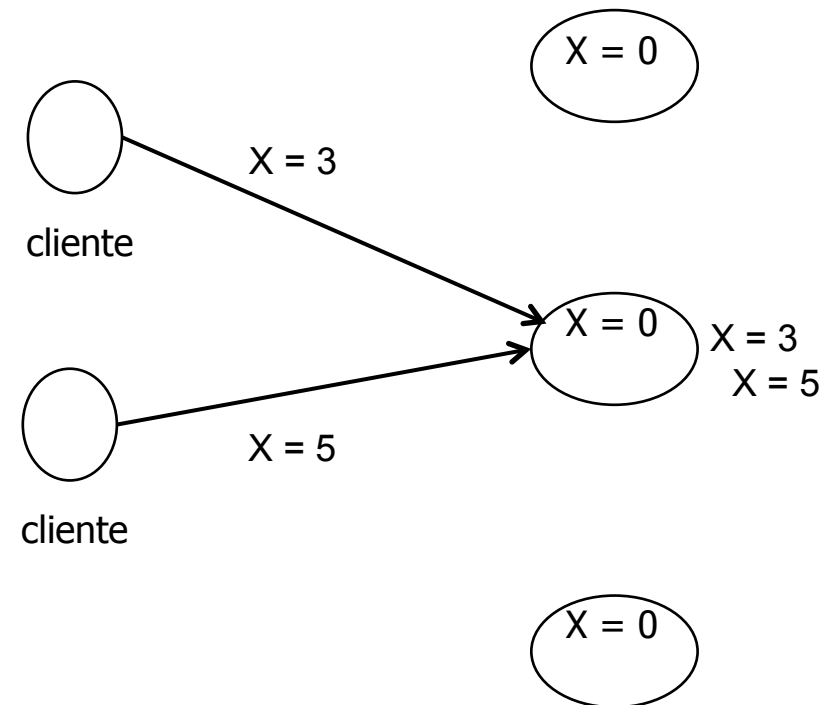
Escritas enviadas para o primário



# PRIMÁRIO / SECUNDÁRIO : ESCRITA

Escritas enviadas para o primário

Primário ordena escritas e executa-as ordenadamente



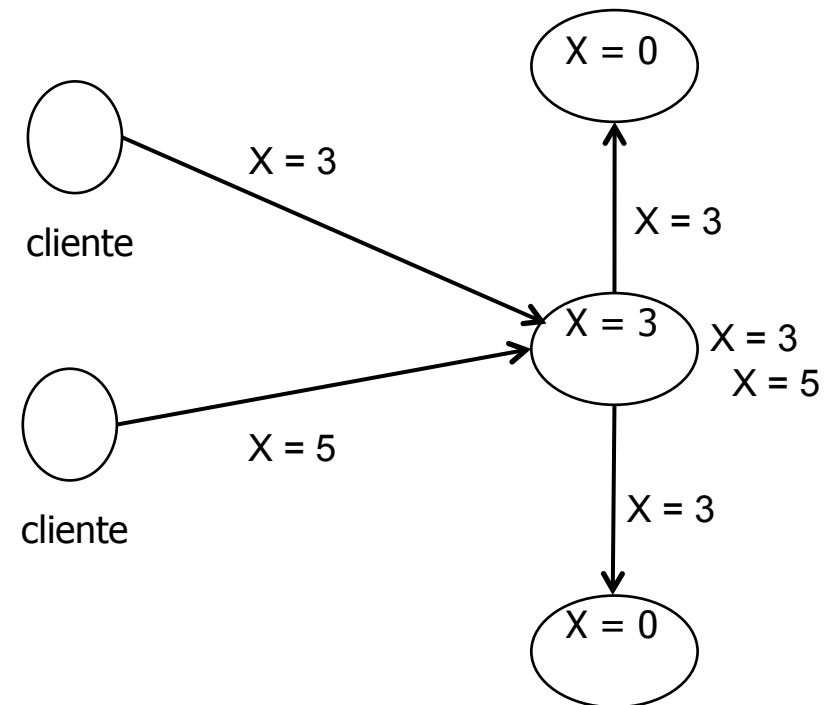
# PRIMÁRIO / SECUNDÁRIO : ESCRITA

Escritas enviadas para o primário

Primário ordena escritas e executa-as ordenadamente

Execução de operação

1. Execução no primário
2. Envio da operação para o secundário



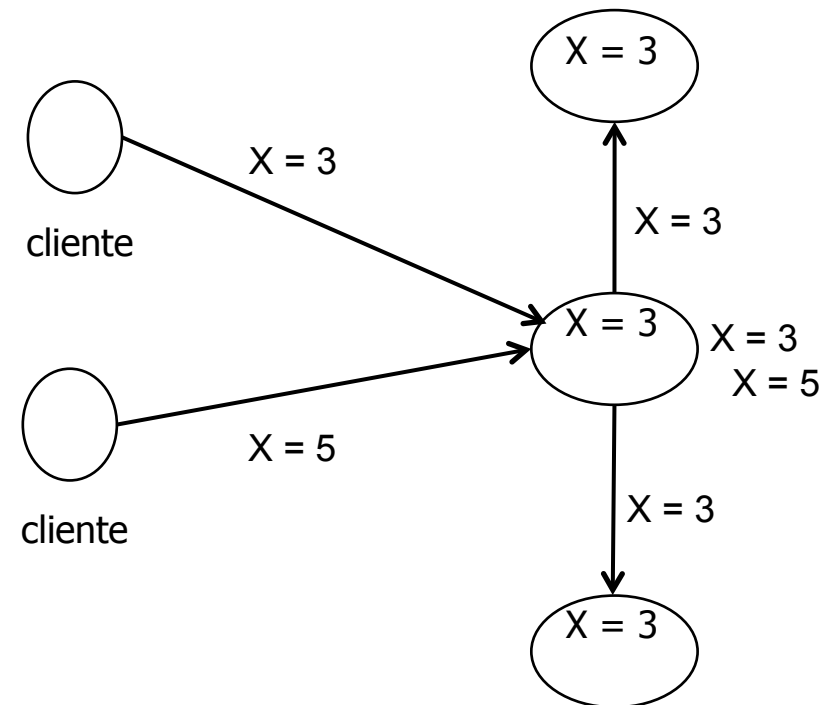
# PRIMÁRIO / SECUNDÁRIO : ESCRITA

Escritas enviadas para o primário

Primário ordena escritas e executa-as ordenadamente

Execução de operação

1. Execução no primário
2. Envio da operação para os secundários
3. Secundário executa operação



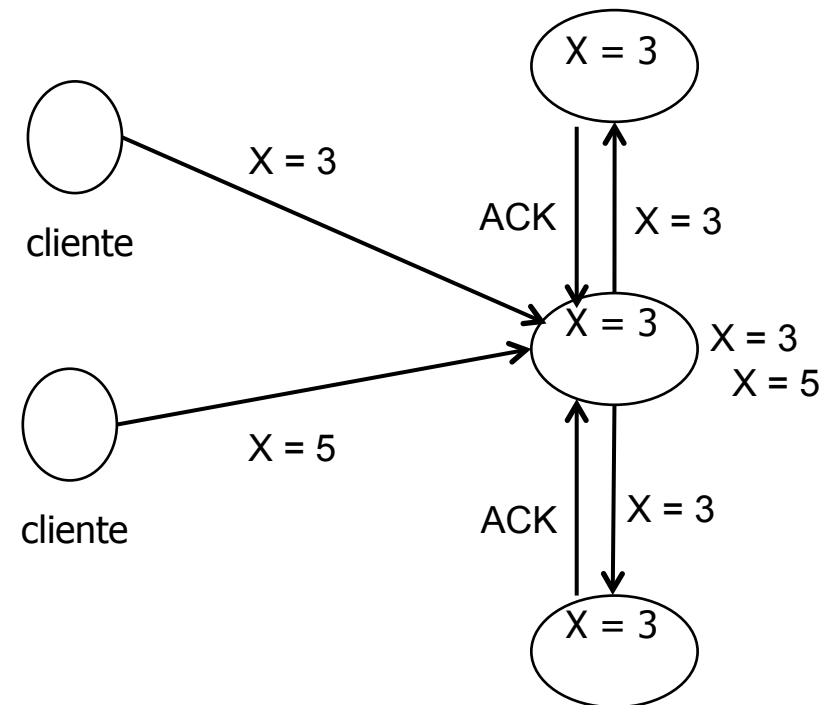
# PRIMÁRIO / SECUNDÁRIO : ESCRITA

Escritas enviadas para o primário

Primário ordena escritas e executa-as ordenadamente

Execução de operação

1. Execução no primário
2. Envio da operação para os secundários
3. Secundário executa operação
4. Secundário envia ACK para primário





# PRIMÁRIO / SECUNDÁRIO : ESCRITA : FALHA DUM SECUNDÁRIO

O que fazer quando um secundário falha?

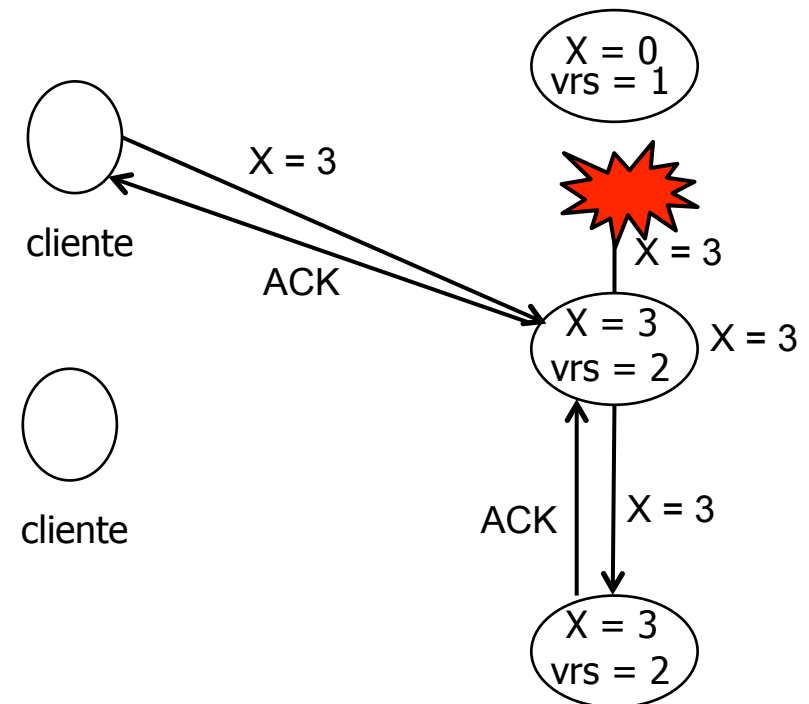
O primário pode devolver ACK ao cliente sem receber a resposta de todos os secundários

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

Operações devem ser numeradas pelo primário (canal FIFO).

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

NOTA: dependendo das operações, receber a última operação pode ser suficiente

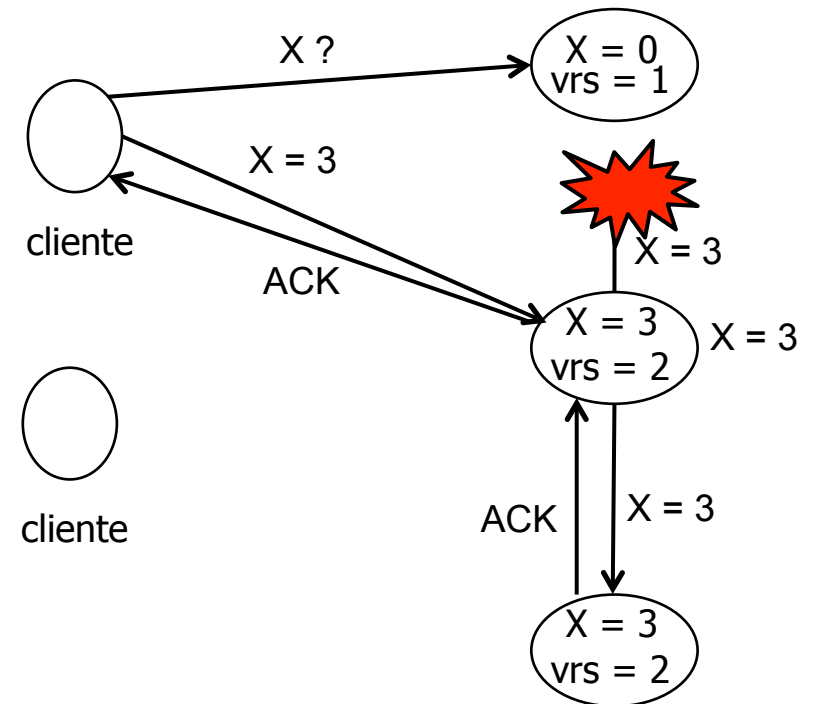


# PRIMÁRIO / SECUNDÁRIO : ESCRITA : FALHA DUM SECUNDÁRIO

O que fazer quando o primário falha?

O primário pode devolver ACK ao cliente sem receber a resposta de todos os secundários

Como é que se garante que cliente não lê versão anterior à sua escrita?



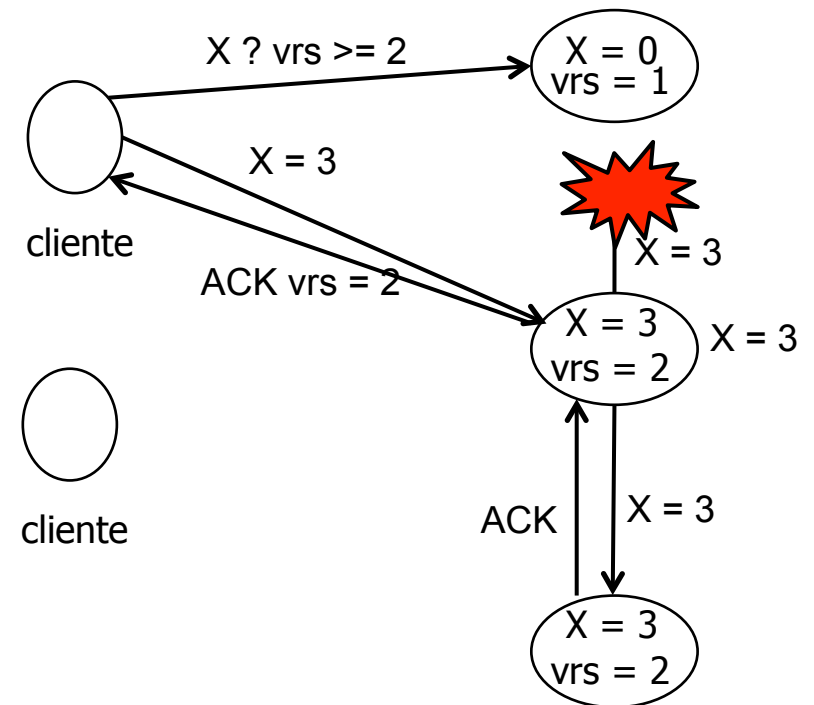
# PRIMÁRIO / SECUNDÁRIO : ESCRITA : FALHA DUM SECUNDÁRIO

O que fazer quando o primário falha?

O primário pode devolver ACK ao cliente sem receber a resposta de todos os secundários

Como é que se garante que cliente não lê versão anterior à sua escrita?

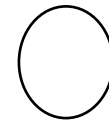
Cliente pode receber versão escrita e enviar essa versão no próximo pedido para que apenas um servidor com uma versão superior responda



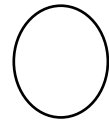
# PRIMÁRIO / SECUNDÁRIO : ESCRITA : FALHA DO PRIMÁRIO

O que fazer quando o primário falha?

1. Um secundário deve ser eleito o novo primário
2. Secundários devem concordar em qual a última operação executada



cliente



cliente

$X = 0$   
 $vrs = 1$

  $X = 3$   
 $vrs = 2$

$X = 3$   
 $vrs = 2$

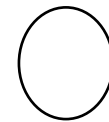
# PRIMÁRIO / SECUNDÁRIO : ESCRITA : FALHA DO PRIMÁRIO

O que fazer quando o primário falha?

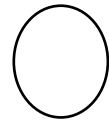
1. Um secundário deve ser eleito o novo primário
2. Secundários devem concordar em qual a última operação executada

Como garantir que apenas um secundário é eleito?

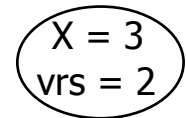
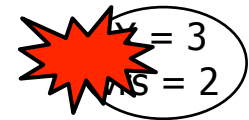
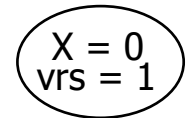
Algoritmo de eleição / sistema de comunicação em grupo  
Detetor de falhas



cliente



cliente



# PRIMÁRIO / SECUNDÁRIO : ESCRITA : ELEIÇÃO DE NOVO PRIMÁRIO : OUTLINE

Consideremos 1 primário + N secundários

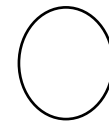
Primário pode continuar a funcionar desde que receba ACK de  $K > N/2$  secundários

Detecção de primário falhado

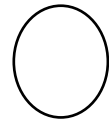
- Quando  $K > N/2$  secundários concordam que primário falhou (e concordam em não receber mais pedidos desse primário)

Eleição de novo primário (solução simples)

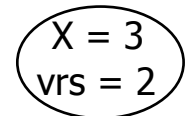
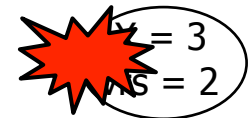
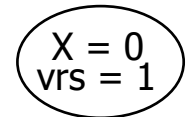
- Servidores estão ordenados
- Quando o primário falha, o próximo servidor será o novo primário



cliente



cliente



Problemas com esta solução simples?

# CODA

Sistema de ficheiros distribuído, descendente do AFS

Servidores replicam volumes (hierarquia de directórios): réplicas principais

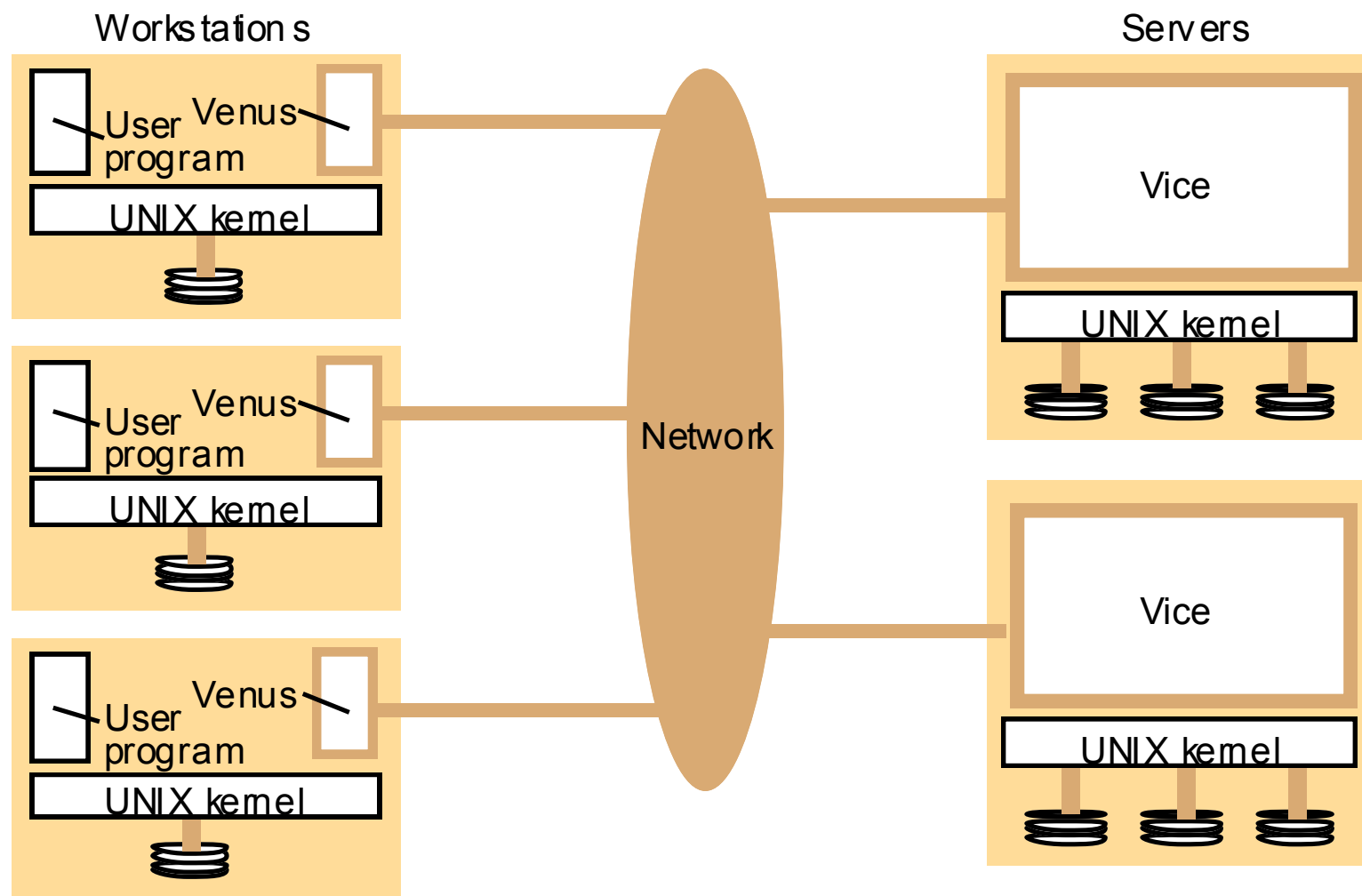
Clientes replicam subconjunto de ficheiros: réplica secundária (ou *cache*)

Ficheiros são replicados completamente

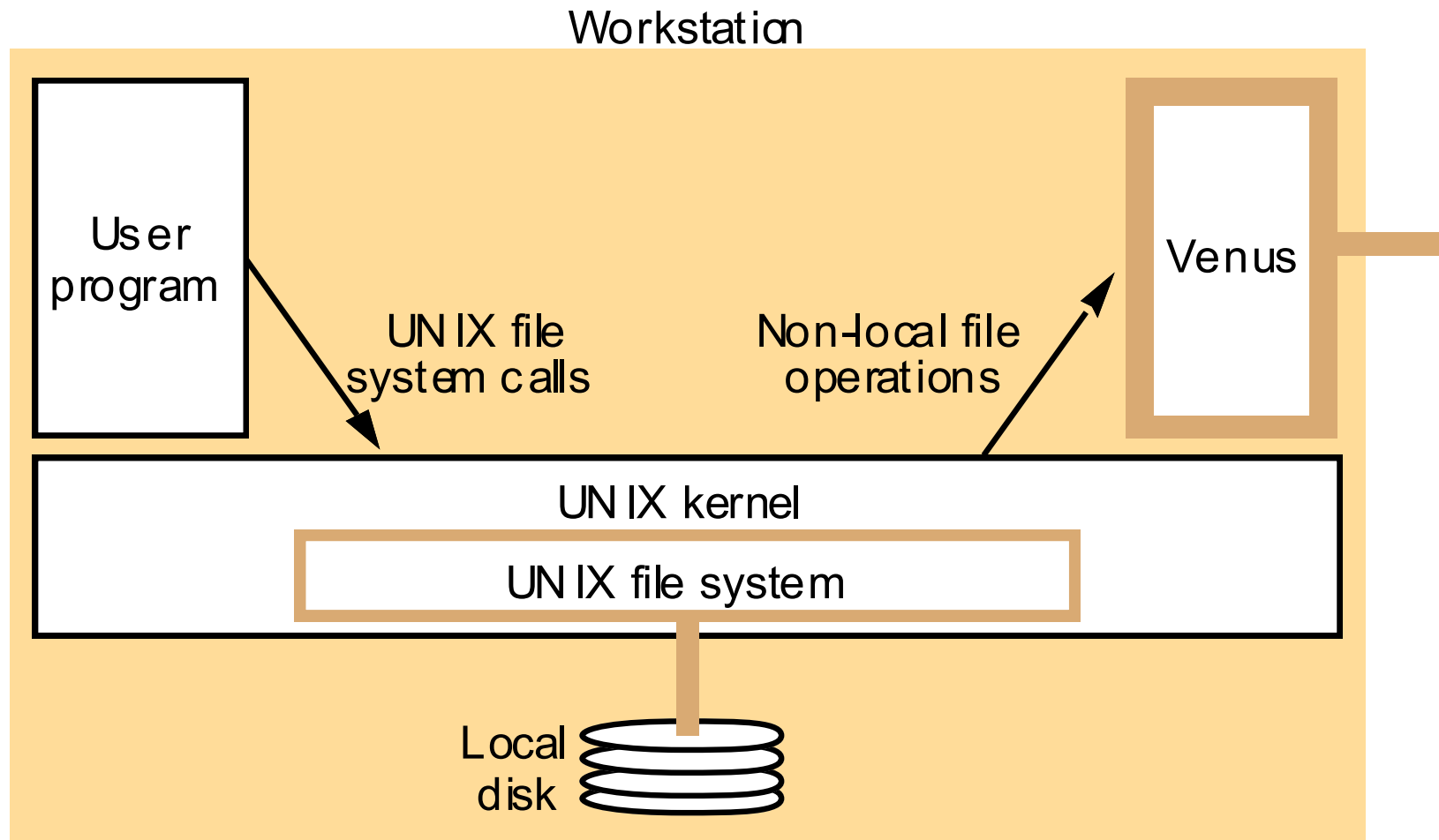
Utiliza aproximação “callback promise”

Modificações no servidor notificadas aos clientes que replicam ficheiros

# ESTRUTURA DO SISTEMA CODA



# FUNCIONAMENTO DOS SYSTEM CALLS



# REPLICAÇÃO NO SISTEMA CODA

## Aproximação optimista

Permite modificação com partições na rede e durante períodos de desconexão

Cada cliente mantém dois conjuntos relativamente a cada volume :

VSG – conjunto de servidores que replicam o volume

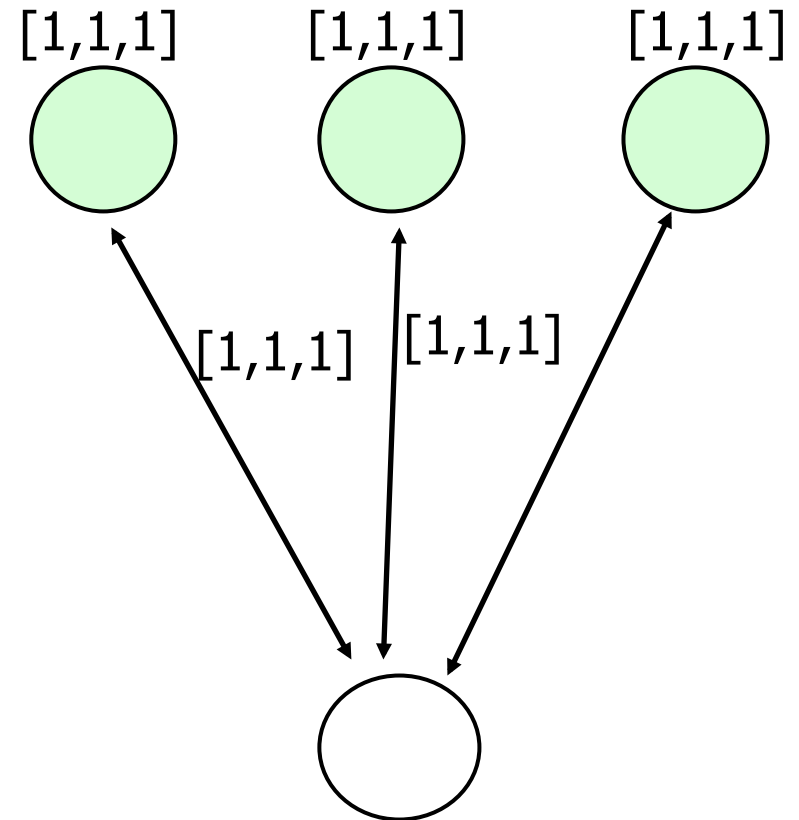
AVSG – sub-conjunto dos servidores VSG activos

Cada ficheiro tem associado um vector-versão (semelhante a um relógio vectorial),  
que identifica a sua versão

No vector-versão existe uma entrada para cada servidor do VSG

# REPLICAÇÃO NO SISTEMA CODA: LEITURA

Um cliente obtém uma cópia de um ficheiro a partir de um elemento do AVSG, verificando com todos qual o que tem a versão mais actual

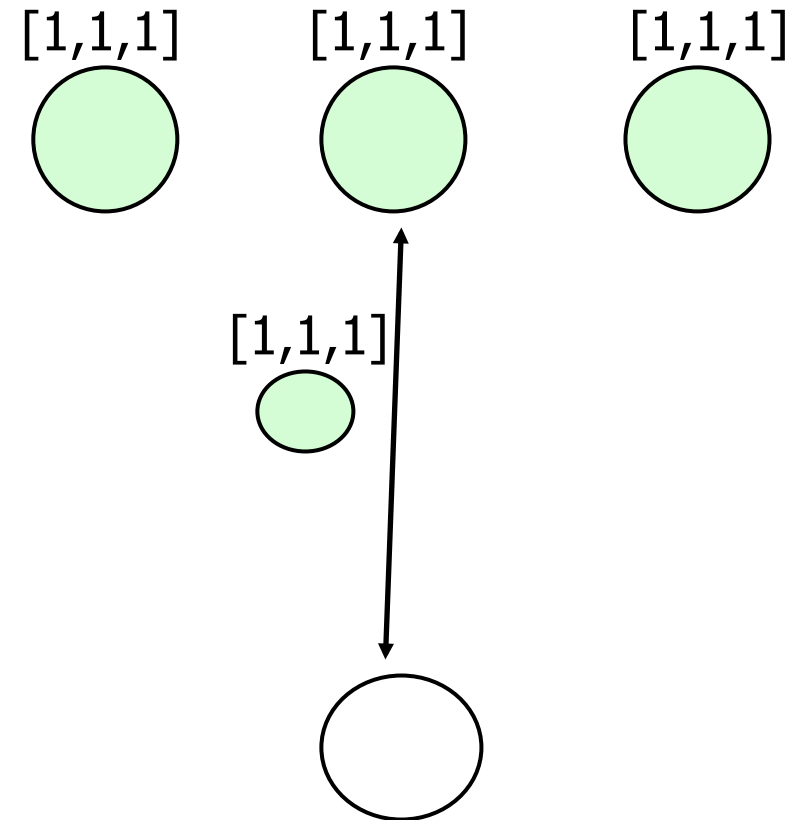


# REPLICAÇÃO NO SISTEMA CODA: LEITURA

Um cliente obtém uma cópia de um ficheiro a partir de um elemento do AVSG, verificando com todos qual o que tem a versão mais actual

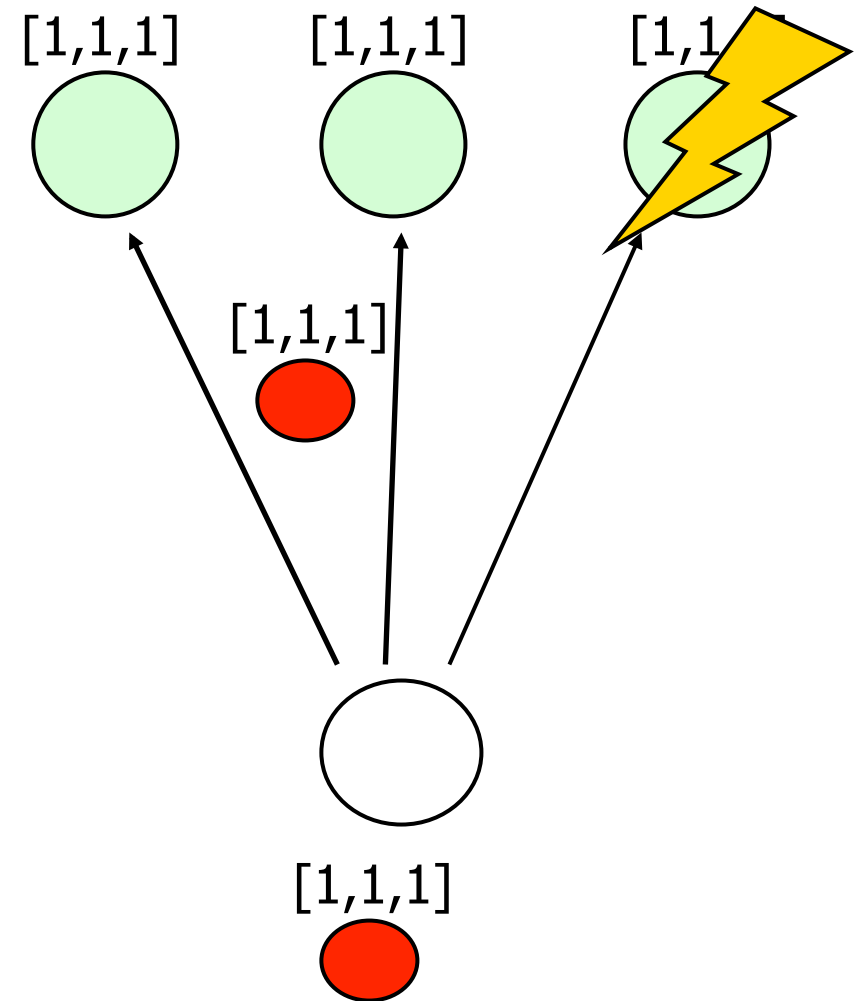
O cliente obtém a cópia do ficheiro de um dos servidores com a versão mais actual. Fica com uma "callback promise" nesse servidor (e só nesse).

Se existirem diferentes versões, os servidores são informados para se sincronizarem



# REPLICAÇÃO NO SISTEMA CODA: ESCRITA

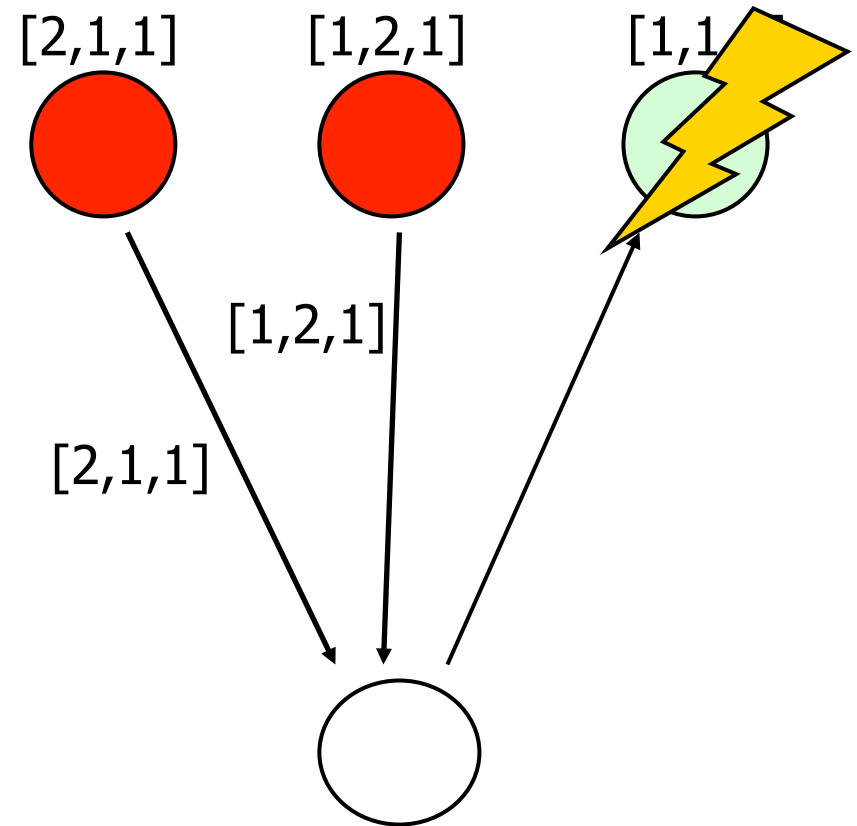
Quando o ficheiro é fechado, o cliente escreve a nova versão em todos os servidores do AVSG



# REPLICAÇÃO NO SISTEMA CODA: ESCRITA

Quando o ficheiro é fechado, o cliente escreve a nova versão em todos os servidores do AVSG

Os servidores actualizam o seu vector-versão e enviam-no ao cliente

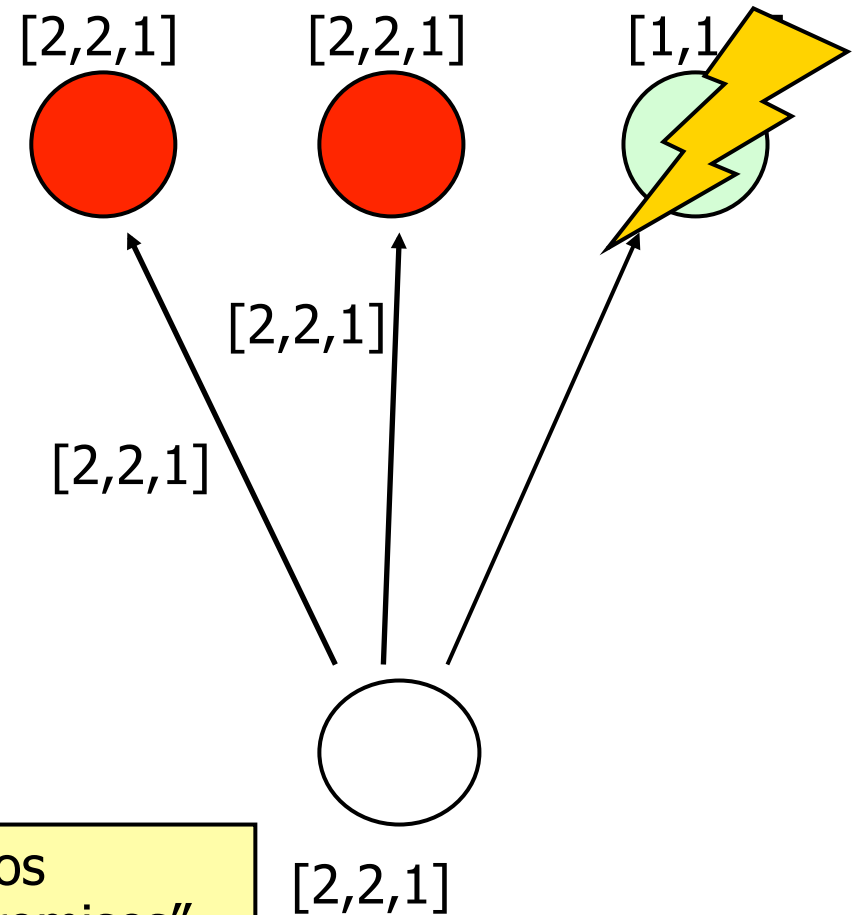


# REPLICAÇÃO NO SISTEMA CODA: ESCRITA

Quando o ficheiro é fechado, o cliente escreve a nova versão em todos os servidores do AVSG

Os servidores actualizam o seu vector-versão e enviam-no ao cliente

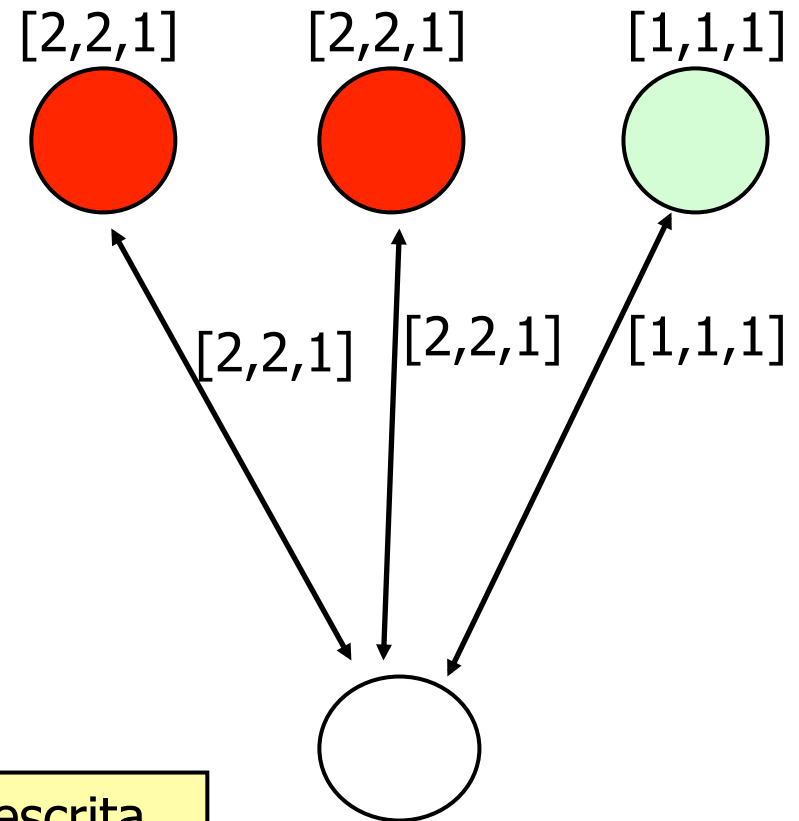
O cliente combina os vectores versão e envia-os aos servidores, que actualizam o seu vector-versão



Os servidores notificam os clientes com "callback promises" das alterações nos ficheiros

# DETECÇÃO DA INCONSISTÊNCIA: LEITURA

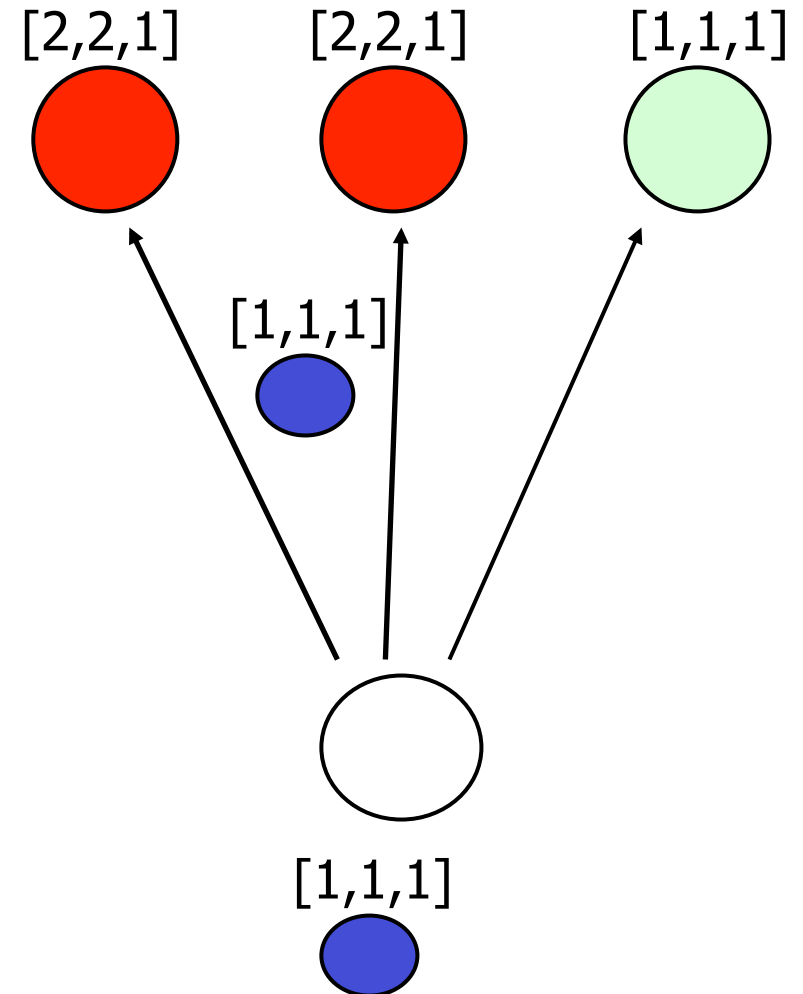
- Se o vector-versão dos vários servidores é diferente
- Dados dois vectores-versão,  $v1$  e  $v2$ :
  - Se  $v2 > v1$ ,  $v2$  é uma versão mais actual
  - Se  $v2 \parallel v1$ , existiram escritas concorrentes



Como poderia surgir uma escrita concorrente?

# DETECÇÃO DA INCONSISTÊNCIA: ESCRITA

Se quando se envia uma escrita, o vector-versão da versão modificada for inferior ao da réplica actual é porque existiu uma modificação concorrente (conflito)



# TRATAMENTO DE CONFLITOS

## Resolução manual (inicial)

Quando existem modificações concorrentes a um ficheiro, os ficheiros são marcados como em conflito e o acesso é impedido até o conflito ser resolvido por um utilizador

## Resolução automática

Podem-se definir programas que, em caso de conflito, são executados para gerar o estado final do ficheiro a partir das diferentes versões

Modificações concorrentes em directorias são resolvidas automaticamente (usando as operações)

# COERÊNCIA DA CACHE

A coerência da cache é mantida pelo mecanismo de “callback promise”

Um cliente invalida uma “callback promise” se o conjunto AVSG aumenta (i.e., se um servidor passa a estar acessível). Porquê?

Cada servidor mantém, além do vector versão para cada ficheiro, um vector versão para um volume. Um cliente invalida uma “callback promise” se, ao contactar os servidores, verifica que eles têm diferentes vectores versão de volume. Porquê?

Qual o interesse deste vector-versão?

# 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