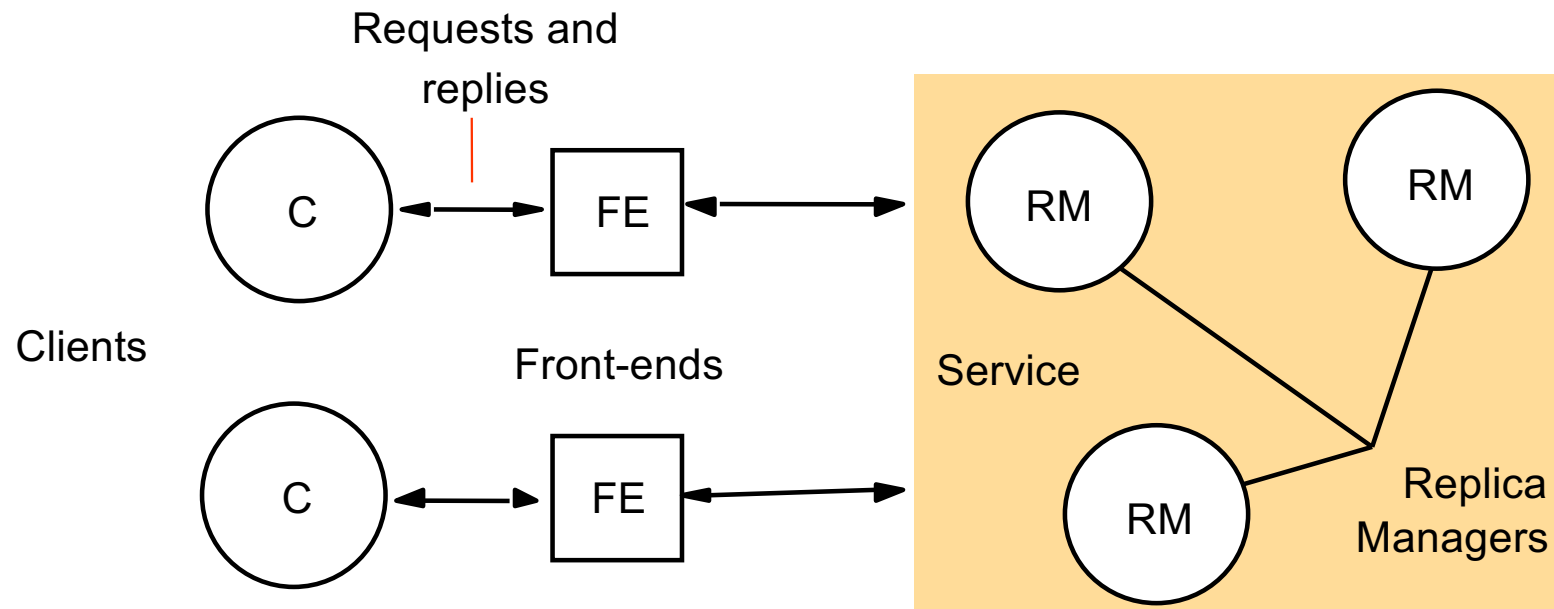


Breve introdução aos sistemas replicados

Replicação

- Conceito simples:
 - Manter **cópias** dos dados e do software do serviço em vários computadores



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
 - Caso extremo: cópia na própria máquina do cliente (*cache*)
 - Algumas operações podem ser executadas apenas sobre algumas das cópias
 - Distribui-se carga, logo maior escalabilidade

Coerência

- **Coerência**

- Idealmente, um cliente que leia de uma das réplicas deve sempre ler **o valor mais atual**
 - Mesmo que a escrita mais recente tenha sido solicitada sobre outra réplica

- Um critério de coerência: linearizabilidade

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

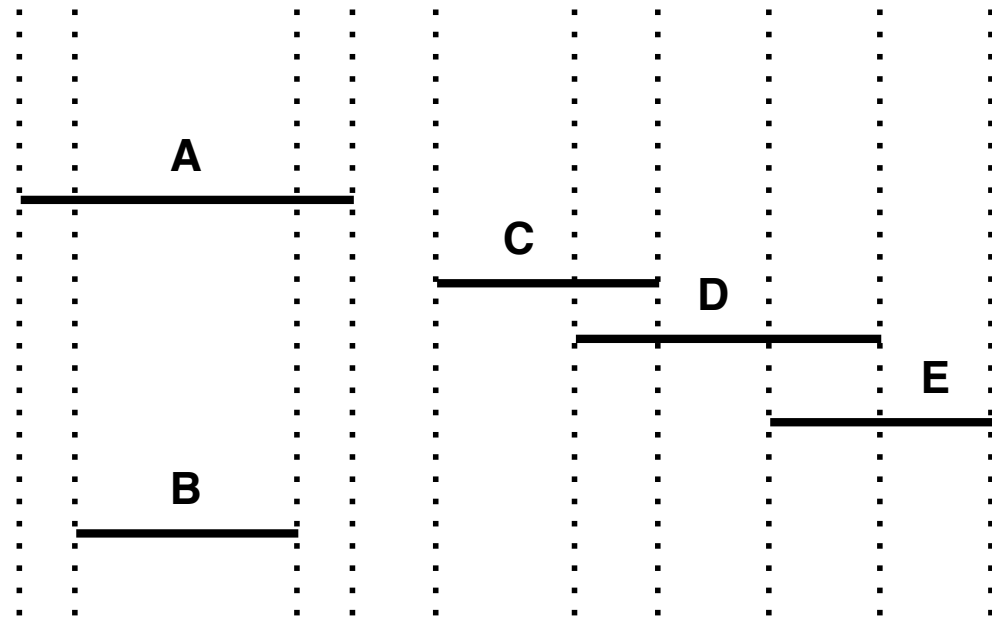
A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects techniques from the sequential domain. Linearizability provides the illusion that each process takes effect instantaneously at some point between its operations. This paper compares it to other correctness conditions, and

Nota prévia sobre ordenar operações

- As operações sobre um sistema replicado não são instantâneas
 - Uma operação é invocada por um cliente, executada e mais tarde o cliente recebe a resposta
- Se uma operação X começa depois de outra operação Y acabar, a operação X ocorre **depois** de Y
- Se uma operação X começa antes de outra operação Y acabar, a operação X é **concorrente** com Y

Ordenar operações

- A concorrente com B
- A anterior a C
- B anterior a C
- C concorrente com D
- C anterior a E
- D concorrente com E



Linearizabilidade

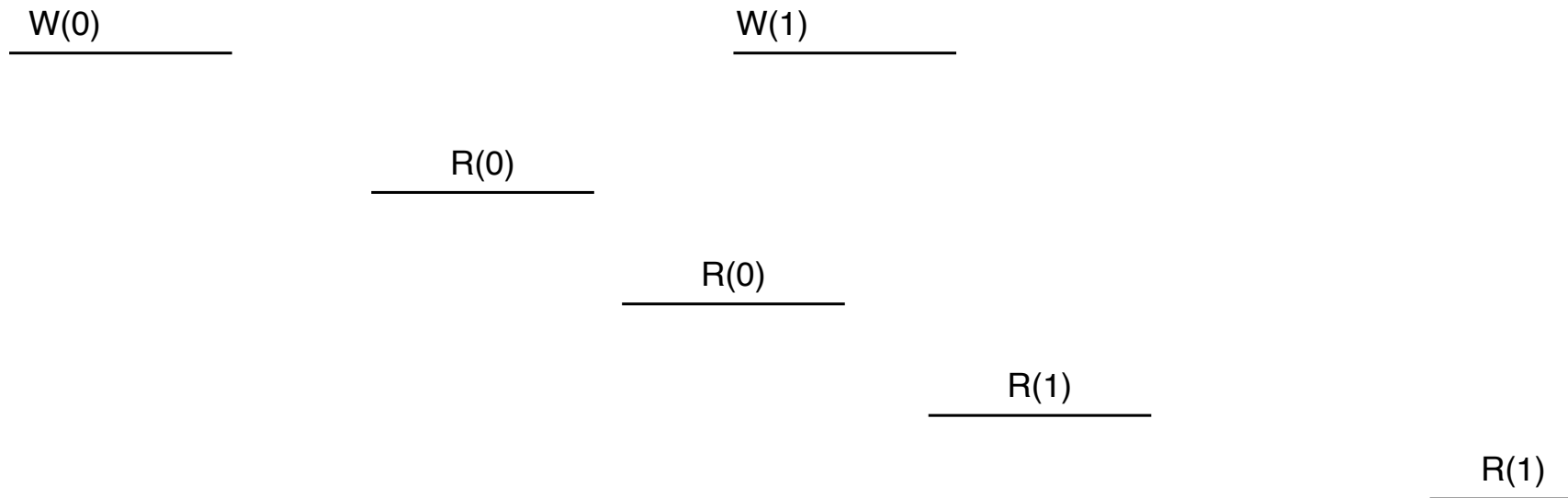
Um sistema replicado diz-se **linearizável** sse (se e só se):

1. Existe uma **serialização virtual** que respeita o **tempo real** em que as operações foram invocadas, isto é:

Se $op1$ ocorre antes de $op2$ (em tempo real), então $op1$ tem de aparecer antes de $op2$ na serialização virtual

- Nota: se $op1$ e $op2$ concorrentes, a serialização virtual pode ordená-las arbitrariamente
2. A execução observada por cada cliente é coerente com essa serialização virtual (para todos os clientes)
- Isto é, os valores retornados pelas leituras feitas por cada cliente refletem as operações anteriores na serialização virtual

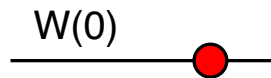
Exemplo I: clientes que escrevem sobre um inteiro replicado



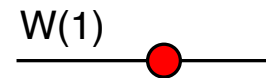
Esta execução é linearizável?

Exemplo I: clientes que escrevem sobre um inteiro replicado

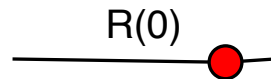
W(0)

A horizontal line with a red dot in the middle, representing a write operation W(0).

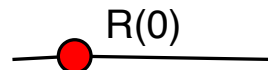
W(1)

A horizontal line with a red dot in the middle, representing a write operation W(1).

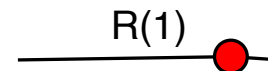
R(0)

A horizontal line with a red dot in the middle, representing a read operation R(0).

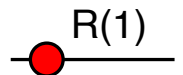
R(0)

A horizontal line with a red dot in the middle, representing a read operation R(0).

R(1)

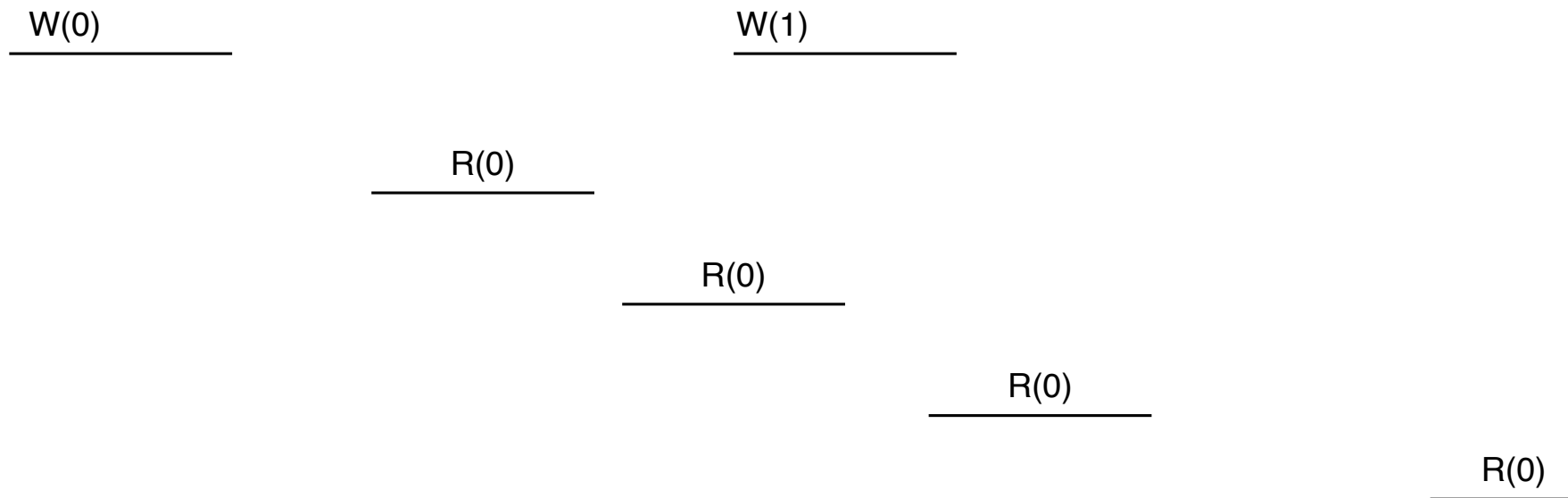
A horizontal line with a red dot in the middle, representing a read operation R(1).

R(1)

A horizontal line with a red dot in the middle, representing a read operation R(1).

Sim, é linearizável

Exemplo II



Esta execução é linearizável?

Exemplo III

W(0)

W(1)

R(0)

R(1)

R(0)

R(1)

Esta execução é linearizável?

Esta aula: como podemos replicar **estruturas de dados específicas**, registo e espaço de tuplos?

Mais tarde, estudaremos como replicar objetos mais **genéricos**

Registos partilhados
(e replicados)

Registos

- Duas operações:
 - Escrita
 - Leitura
- Uma nova escrita substitui o valor da escrita anterior
 - Isto difere do caso em que os objectos aceitam operações arbitrárias (por exemplo, incrementar duas vezes é diferente de incrementar apenas uma vez)
- Múltiplos clientes podem ler do registo, mas só um cliente pode escrever
 - Ou seja, as escritas são totalmente ordenadas
 - *Há algoritmos que suportam múltiplos escritores, mas ficam fora da cadeira*

Registros

- Lamport definiu três modelos de coerência para registros:

- Atomic
- Regular
- Safe



Equivalente a linearizabilidade quando aplicada a registros

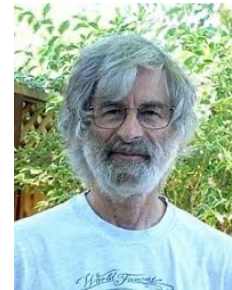
Registos

8

On Interprocess Communication

Leslie Lamport

December 25, 1985



Registos “Atomic”

- Equivalente a linearizabilidade quando aplicada a registos
- *O resultado da execução é equivalente ao resultado de uma execução em que todas as escritas e leituras ocorrem instantaneamente num ponto entre o início e o fim da operação*

Registos “Regular”

- Se uma leitura não for concorrente com uma escrita, lê o último valor escrito.
- Se uma leitura for conconcorrente com uma escrita ou retorna o valor anterior ou o valor que está a ser escrito
- Porque não é tão forte como registo atómico (linearizável)?
- R: Enquanto uma escrita está a decorrer, permite que leituras seguidas (uma a seguir à outra) leiam sequência incoerente de valores (primeiro ler o novo valor, depois ler o valor antigo)

Registos “Safe”

- Se uma leitura não for concorrente com uma escrita, lê o último valor escrito.
- Se uma leitura for conconcorrente com uma escrita pode retornar um valor arbitrário.

Registro “unsafe”

W(0)

W(1)

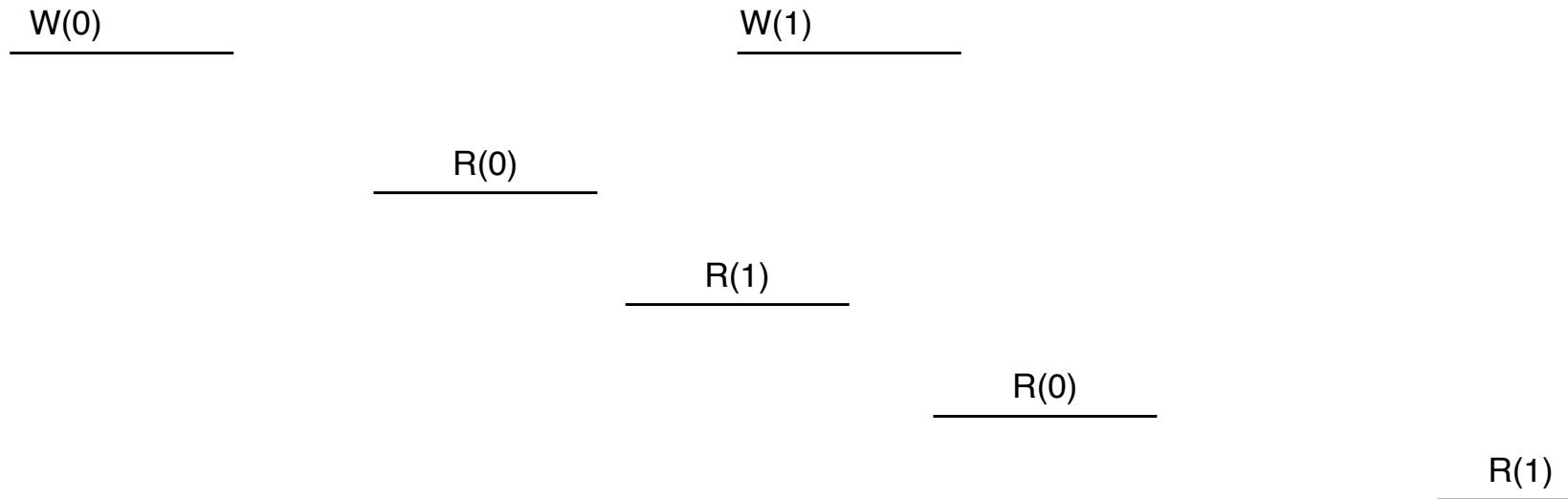
R(0)

R(0)

R(0)

R(0)

Registo “Regular” (mas não “atomic”)



Registro “Atomic”

W(0)

W(1)

R(0)

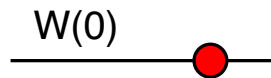
R(0)

R(1)

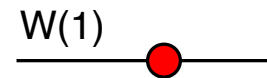
R(1)

Registro "Atomic"

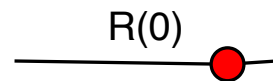
W(0)

A horizontal line with a red dot in the center, representing a write operation to memory address 0.

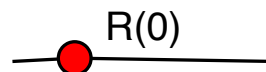
W(1)

A horizontal line with a red dot in the center, representing a write operation to memory address 1.

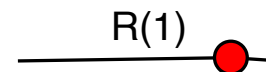
R(0)

A horizontal line with a red dot in the center, representing a read operation from memory address 0.

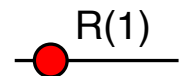
R(0)

A horizontal line with a red dot in the center, representing a read operation from memory address 0.

R(1)

A horizontal line with a red dot in the center, representing a read operation from memory address 1.

R(1)

A horizontal line with a red dot in the center, representing a read operation from memory address 1.

Como concretizar registos distribuídos

- Cada processo mantém uma cópia do registo
- Cada registo mantém um tuplo <valor, versão>
- Para executar uma escrita ou uma leitura, cada processo troca mensagens com os outros processos
- É possível fazer isto de forma tolerante a faltas e não bloqueante!

ABD



Sharing Memory Robustly in Message-Passing Systems

Hagit Attiya¹

Amotz Bar-Noy²

Danny Dolev³

February 16, 1990

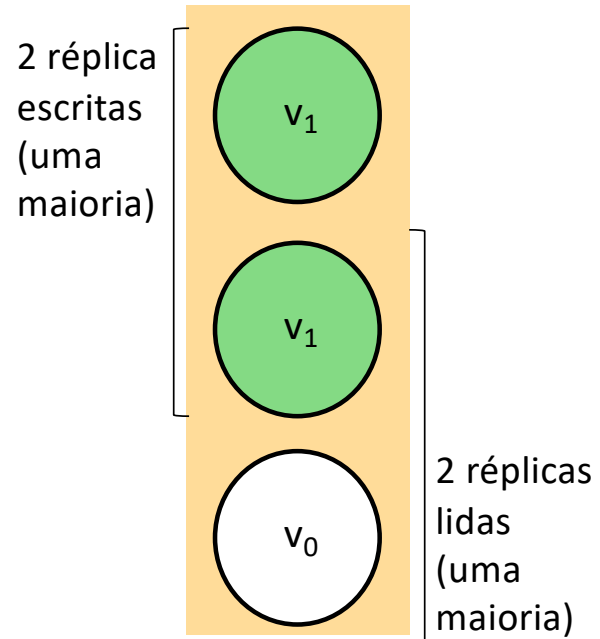
Registo Regular (só com um escritor)

- Escrita:
 - O escritor incrementa o número de versão e envia o tuplo <valor, versão> para todos os processo.
 - Ao receber esta mensagem, os outros processos actualizam a sua cópia do registo (se a versão for superior à que possuem) e enviam uma confirmação o escritor.
 - A operação de escrita considera-se terminada quando o escritor receber resposta de uma maioria

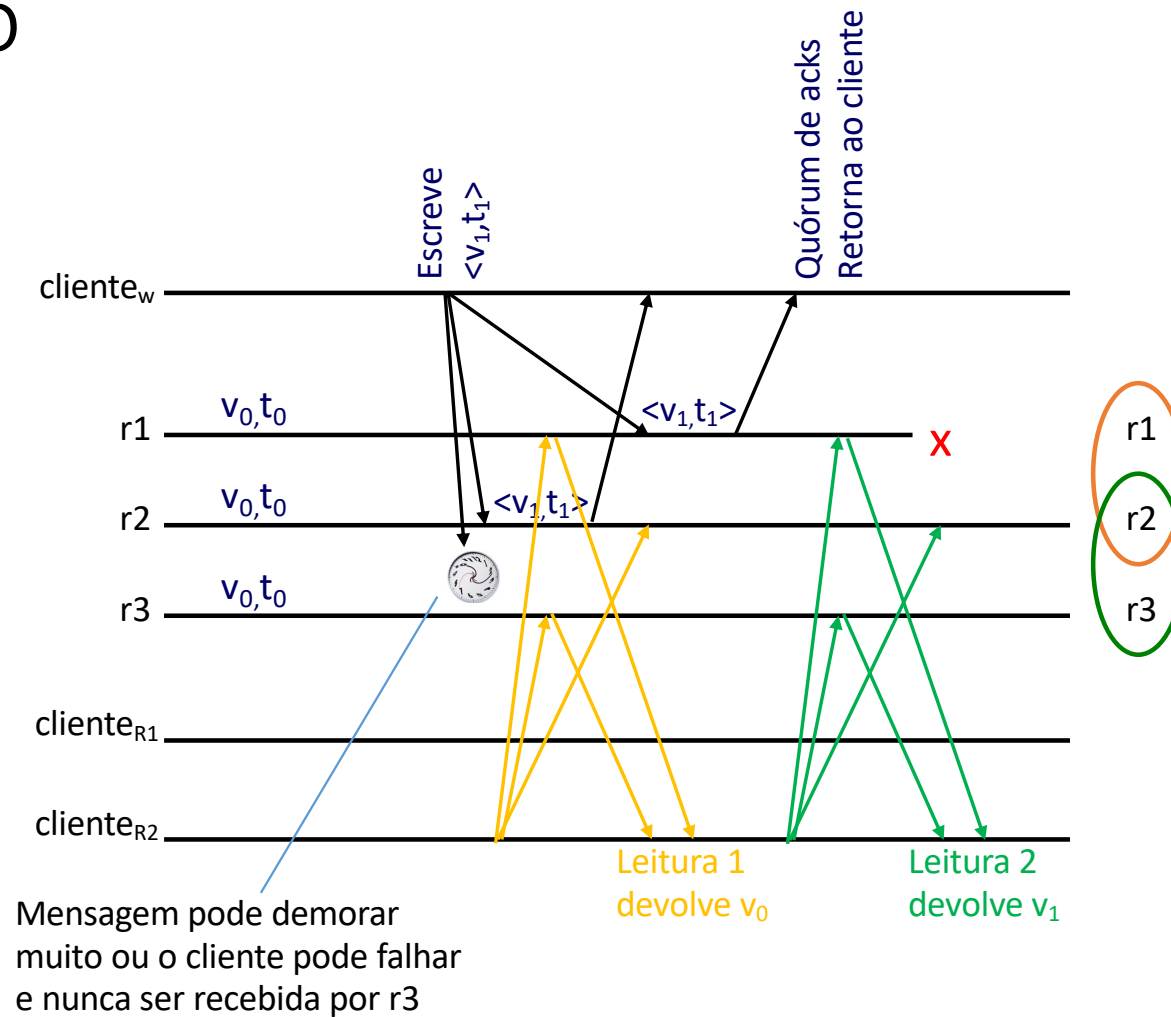
Registo Regular (só com um escritor)

- Leitura:
 - O leitor envia uma mensagem a todos os processos solicitando o tuplo mais recente
 - Cada processo envia o seu tuplo <valor, versão>
 - Após receber resposta de uma maioria, o leitor retorna o valor com a versão mais recente (e actualiza o seu próprio tuplo, caso necessário)

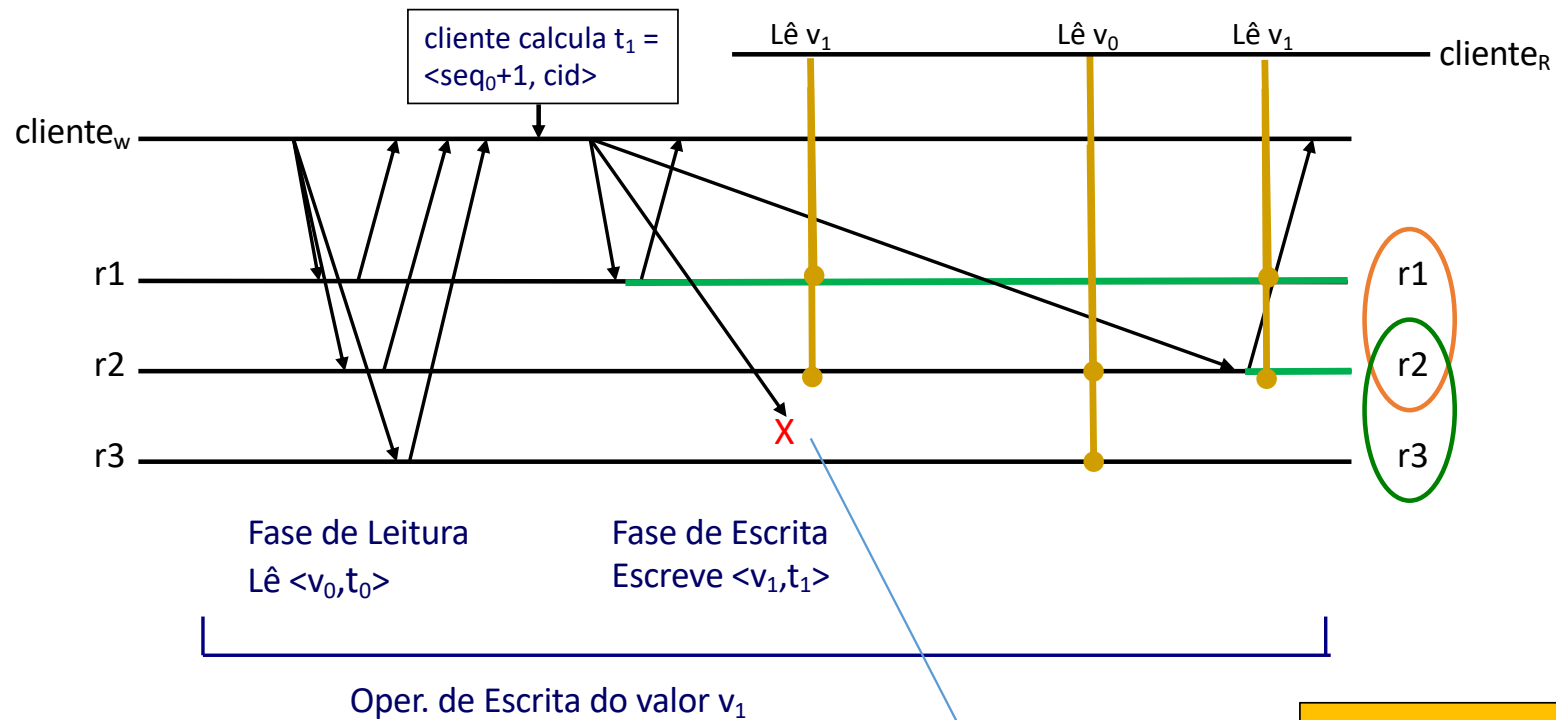
Intuição do algoritmo



Exemplo



Outro exemplo



Mensagem pode demorar muito ou o cliente pode falhar e nunca ser recebida por r3 – **escrita incompleta**

Registo atómico?

Registo regular => registo atómico?

- É possível concretizar um registo atómico de forma não bloqueante, isto é, sem impedir as leituras durante uma escrita?

Registro atômico

- Escrita
 - Semelhante ao anterior com a seguinte diferença.
- Leitura:
 - Executa o algoritmo de leitura anterior mas não retorna o valor
 - Executa o algoritmo de escrita, usando o valor lido
 - Apenas retorna o valor lido após a escrita ter terminado

Espaços de Tuplos

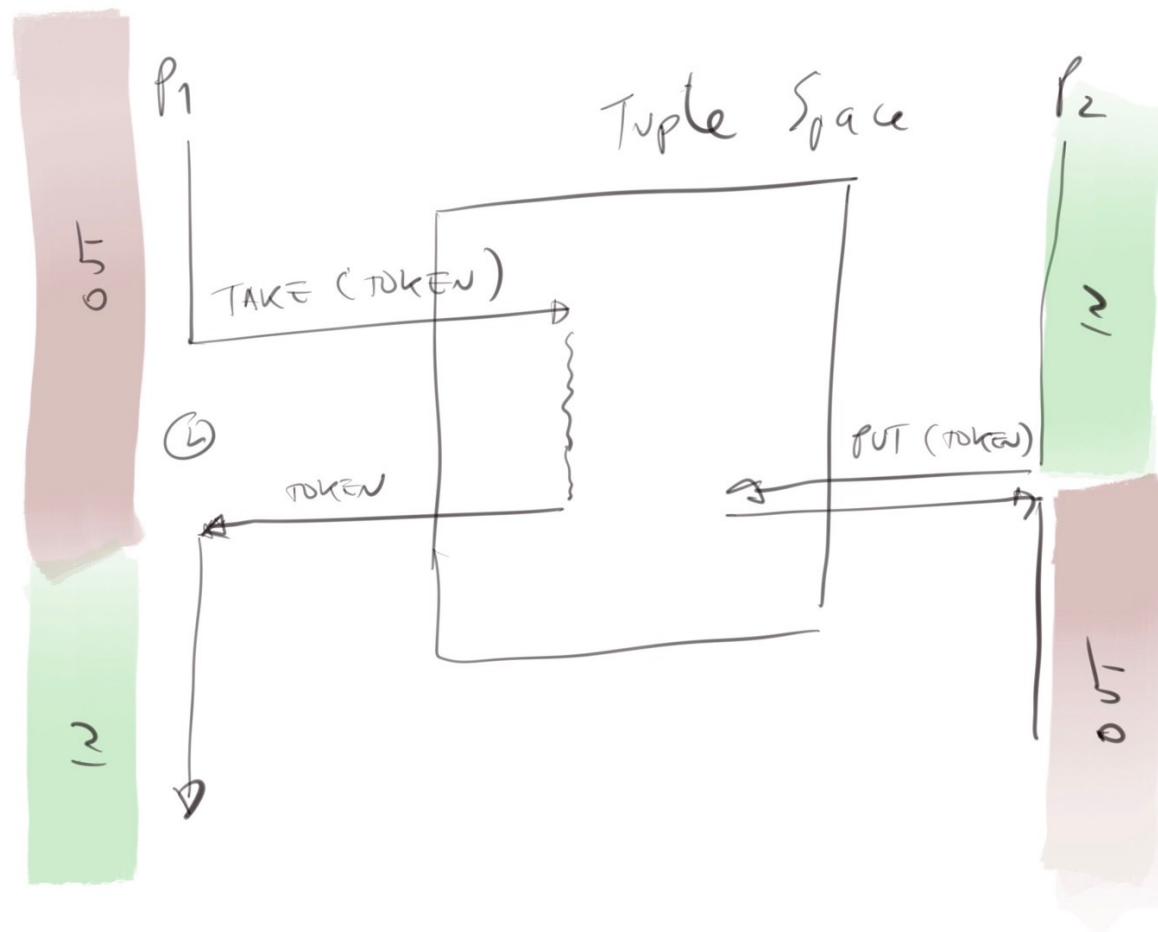
Espaço de Tuplos “Linda”

- Espaço partilhado que contêm um conjunto de tuplos
- Exemplos de tuplos:
 - <“token”>, <“moeda”, “luis”, 1€>, <“moeda”, “luis”, 2€>
- Três operações
 - Put: (originalmente out) coloca um tuplo no espaço
 - Read (originalmente rd):: lê um tuplo do espaço
 - Take (originalmente in): remove um tuplo do espaço (bloqueante, se não existir o tuplo)
 - Read e Take aceitam “wildcards”, p.ex., Take(<“moeda”, “luis”, *>) tanto pode retirar <“moeda”, “luis”, 1€> como <“moeda”, “luis”, 2€>

Espaço de Tuplos “Linda”

- O Take, sendo bloqueante, permite sincronizar processos
- Por exemplo, a exclusão mútua pode ser concretizada através de um tuplo <token>, que um processo remove do espaço antes de aceder ao recurso e que volta a colocar no espaço depois de aceder ao recurso
- Abstracção muito elegante pois permite partilha de memória e sincronização através de uma interface uniforme.

Sincronização através do espaço de tuplos



Diferença para registos

- Num espaço de tuplos, podem existir várias cópias do mesmo tuplo
- O equivalente a uma escrita é feito através de um Take seguido de um Put
- O Take permite fazer operações que em memória partilhada requerem uma instrução do tipo compare-and-swap

Nesta aula

- Como manter um espaço de tuplos distribuído e tolerante a falhas?
- Cada processo mantém uma cópia local do espaço de tuplos
- Executam trocas de mensagens para concretizar as operações Put, Read e Take.

Espaço de tuplos distribuído

A Design for a Fault-Tolerant, Distributed Implementation of Linda

Andrew Xu
Oracle Corporation
Belmont, CA 94002

Barbara Liskov
MIT Laboratory for Computer Science
Cambridge, MA 02139



Abstract

A distributed implementation of a parallel system is of interest because it can provide an economical source of concurrency, can be easily scaled to match the needs of particular computations, and can be fault-tolerant. This paper describes a design for such an implementation for the Linda parallel system, in which processes share a memory called the tuple space. Fault-tolerance is achieved by replication: by having more than one copy of the tuple space, some replicas can provide information when others are not accessible due to failures. Our replication technique takes advantage of the semantics of Linda so that processes encounter

extra information; a replica may have contain old information after recovery from a failure. We present a method that solves these problems. Our main contribution is a way of organizing the tuple space and carrying out the operations on it that results in low delay for Linda programs. We refer to this part of our mechanism as the *operations protocol*. In addition we also present a *view change algorithm* that is used to reconfigure the system when failures occur; this algorithm is based on earlier work [6, 7, 18], but is tailored to the needs of the operations protocol.

Our method has some attractive properties. First, replication is completely hidden from the user program; the replicated tuple space appears to be a single entity. Second, the tuple space can tolerate

Algumas observações prévias

- A tolerância a faltas pressupõe um serviço de filiação que mantém o grupo de réplicas
 - Quando uma réplica falha, a filiação do grupo é alterada
- Desta forma, quando o algoritmo espera por “todas” as respostas ou por uma “maioria de respostas”, refere-se à filiação do grupo num dado instante.
- A alteração dinâmica da filiação é um problema complexo por si só e não será debatido nesta aula.

Algumas observações prévias

- Os autores optam por usar UDP (logo a rede pode perder mensagens) e é o próprio algoritmo que faz retransmissão de mensagens
- Solução modular para este problema: usar TCP

Objetivos

- Os autores tentam obter a solução mais eficiente e que responde ao cliente assim que possível
- Apesar disto, asseguram **linearizabilidade**

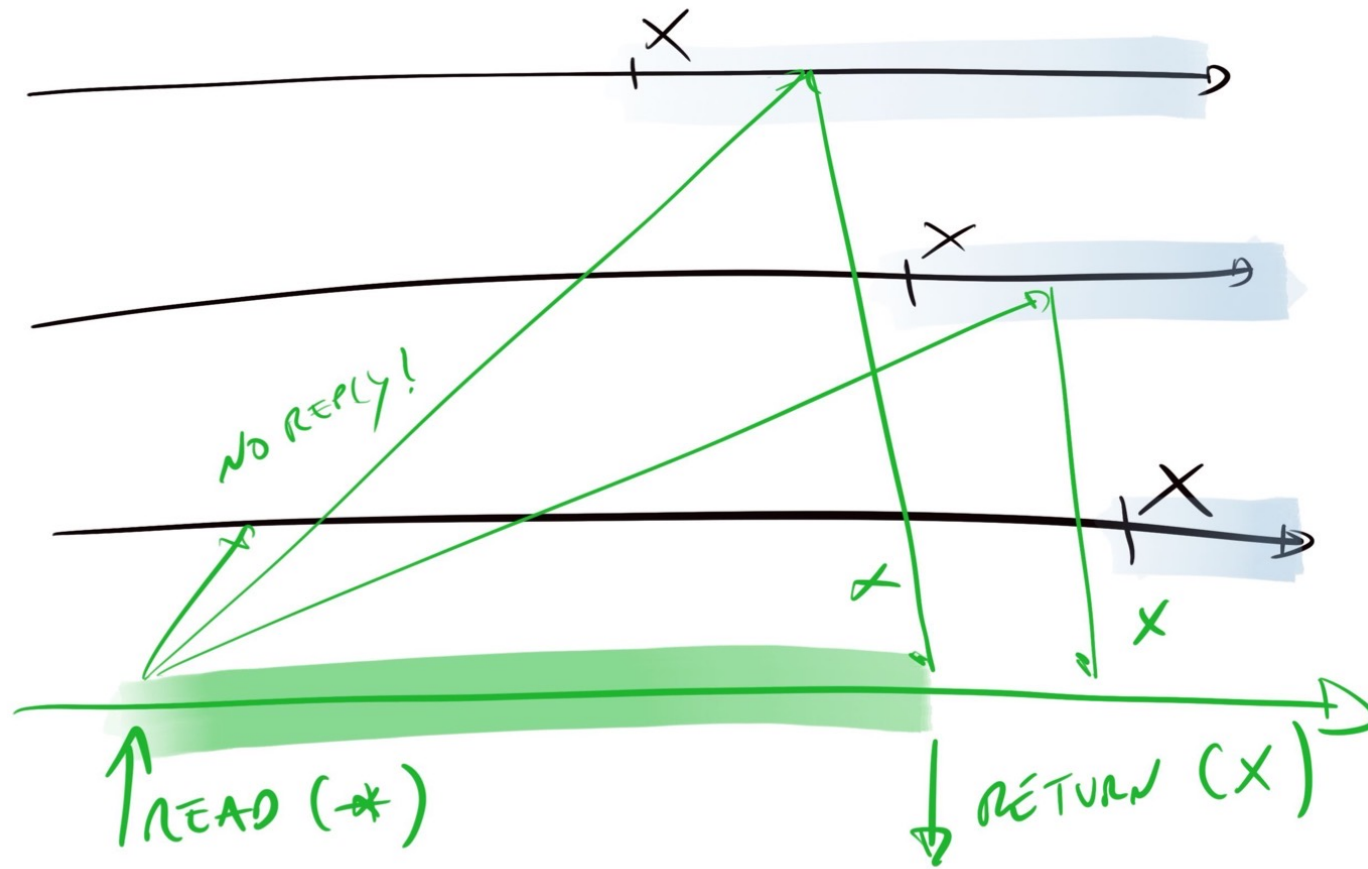
Xu-Liskov

Nota: onde está “write” deve-se ler “put”, para evitar confusão com o “write” dos registros

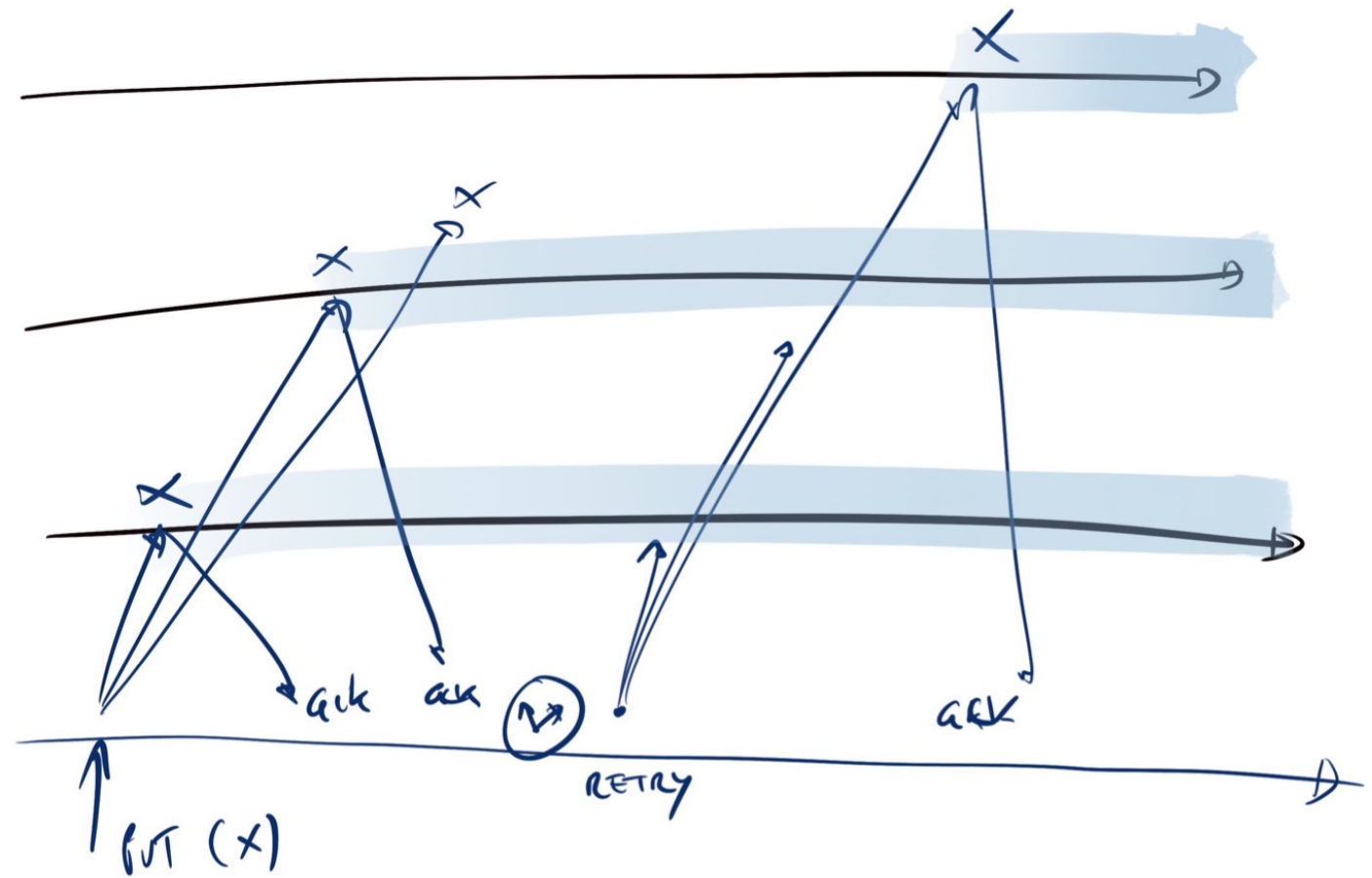
- write*
1. The requesting site multicasts the *write* request to all members of the view;
 2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
 3. Step 1 is repeated until all acknowledgements are received.
- read*
1. The requesting site multicasts the *read* request to all members of the view;
 2. On receiving this request, a member returns a matching tuple to the requestor;
 3. The requestor returns the first matching tuple received as the result of the operation (ignoring others);
 4. Step 1 is repeated until at least one response is received.

Retirado da página 269 do livro da cadeira!

Xu-Liskov



Xu-Liskov



Xu-Liskov

take *Phase 1: Selecting the tuple to be removed*

1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple

1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

Put e Read: discussão

- Imaginemos que o Read não era bloqueante
 - Ou seja, o servidor ou devolvia o tuplo ou “null”
- O sistema continuava a ser linearizável?
- Não, pois este exemplo seria possível:
 - Um cliente executa Put de um tuplo t
 - Um cliente executa Read(t) múltiplas vezes, concorrentemente com o Put
 - Se o leitor receber respostas de réplicas diferentes a cada Read, ele pode ler t e, de seguida, ler “null”

Take: discussão

- Os autores propõem uma solução “custom” que consegue ordem total das operações sobre o mesmo conjunto de tuplos de forma não determinista.
 - No entanto, se todos os processos tentarem fazer Take do mesmo tuplo concorrentemente, pode acontecer que nenhum consiga uma maioria
 - Isto pode repetir-se na próxima tentativa a não ser que se introduza um factor de aleatoriedade no processo (daí a solução não ser determinista)
- Uma solução modular usaria um protocolo de ordem total para este efeito. Estudaremos estes protocolos mais tarde.

Otimizações para minimizar a latência

- O artigo discute várias otimizações que é possível fazer.
- Por exemplo, o artigo sugere que a segunda parte do Take pode ser feita “nos bastidores”, retornando ao cliente e deixando-o fazer novos pedidos em paralelo.

need not wait, instead, the operation can be performed in the background while program execution continues. Similarly, there is no need for the executing worker to be blocked while an In2 is in process.

Nota: esta otimização não faz parte do projeto de SD

Otimizações para minimizar a latência

- Infelizmente, estas optimizações podem levar a anomalias bastante subtis

value from *any* replica. For example, suppose the operations of W_1 and W_2 are executed as follows: W_1 's `In("x", formal x)` and `out("x", $x + 1$)` are executed at R_1 , W_2 's `rd("x", formal u)` is executed at R_1 and returns ("x", 2), and finally, W_2 's `rd("x", formal v)` is executed at R_2 and returns ("x", 1), which is incorrect.

To prevent this problem, we require that requests for an `out` operation not be sent to any replica until previous `In` operations issued by the same worker have been performed at *all* replicas in the current view. Thus, in the above example tuple ("x", 2) cannot exist

Xu-Liskov

