Espaços de Tuplos Replicados

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
- Permite implementar exclusão mútua! Como?
  - 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.

Diferenças 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 memária partilhada requerem uma instrução atómica (e.g., `compare-and-swap`)

Como implementar um espaço de tuplos distribuído?

Usando um algoritmo de replicação

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

Observação prévia I

- 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

Observação prévia II

- 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 registos

write

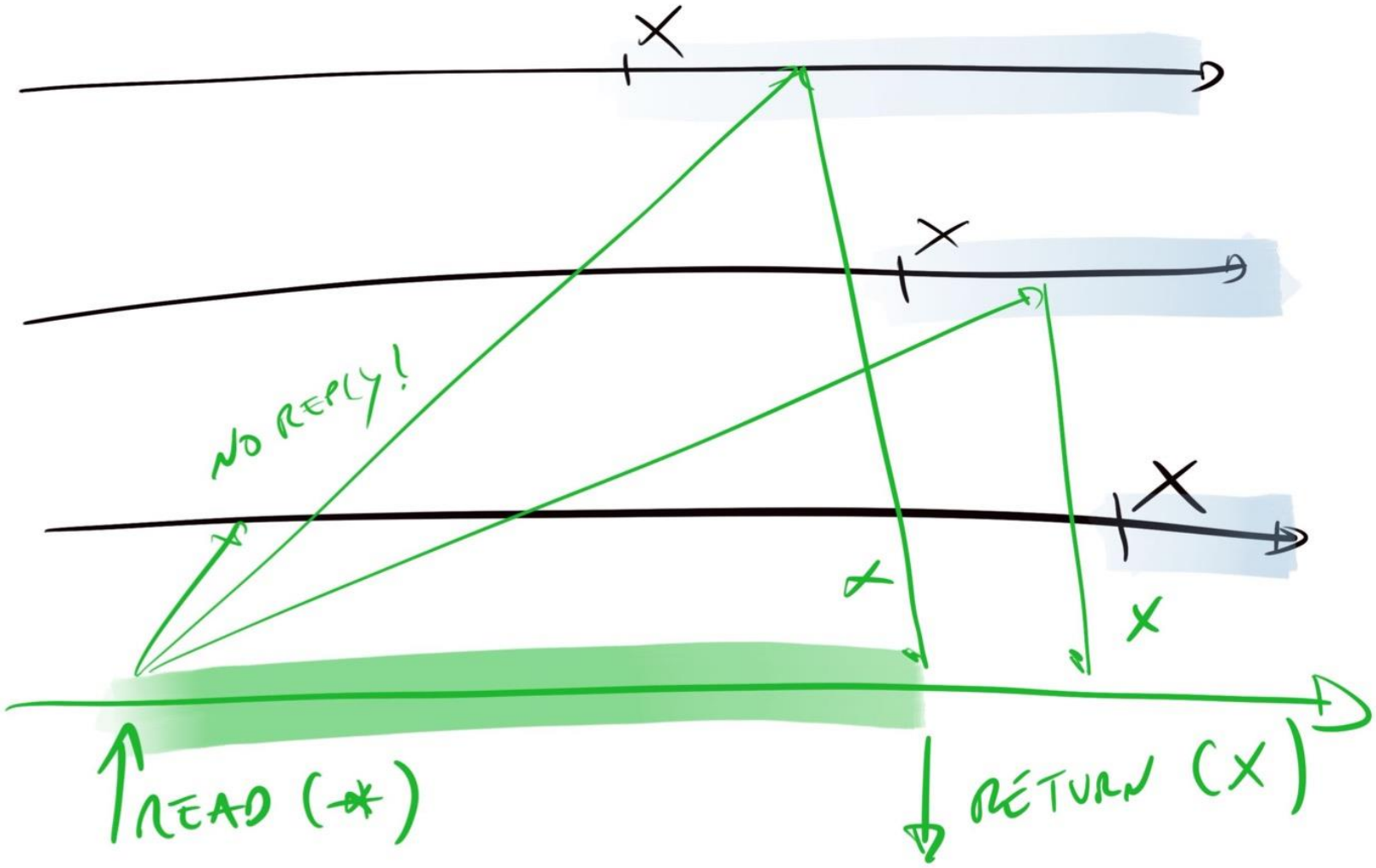
1. The requesting site multicasts the *write* request to all members of the view;
2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

read

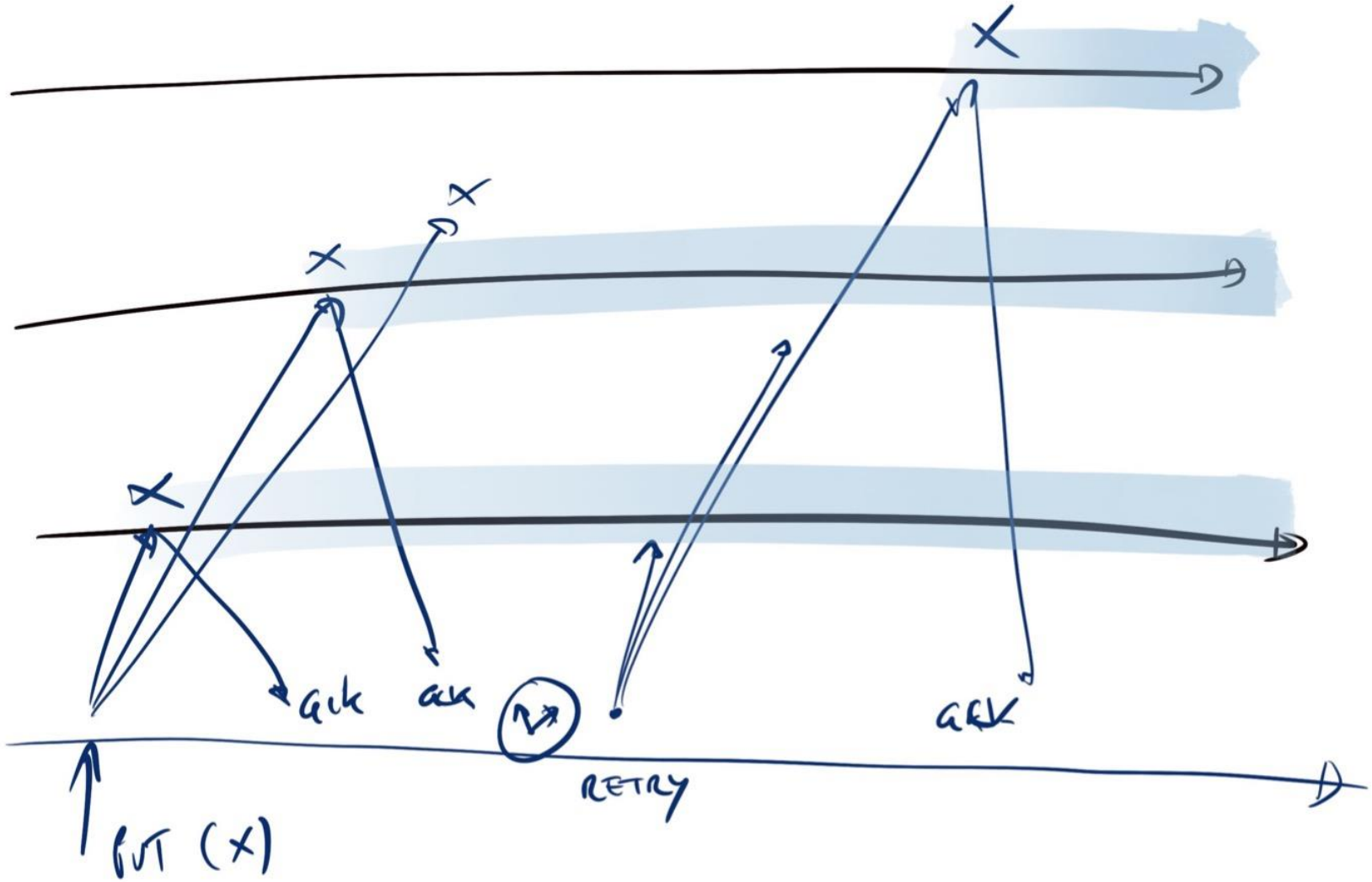
1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others);
4. Step 1 is repeated until at 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 (I)

- Se quisermos implementar isto em GRPC, podemos usar *blocking stubs*?

```
stubs[N] = TupleSpacesServiceGrpc.newBlockingStub(...);
for (s : stubs)
   s.put(...);
return; /* All servers acknowledged the put */
```

- Qual o problema desta implementação?
- Para implementar este algoritmo, precisamos de stubs não bloqueantes (a.k.a. assíncronos)

Put e *Read*: discussão (II)

- Alguma relação com as operações *write* e *read* do algoritmo ABD?
- Sim! Se usarmos o algoritmo ABD com quóruns diferenciados:
  - Quórum de escrita = N
  - Quórum de leitura = 1
- Simplificação: não precisamos de *version number* com espaços de tuplos (mas precisamos com registo)

Put e *Read*: discussão (II)

- 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 latência

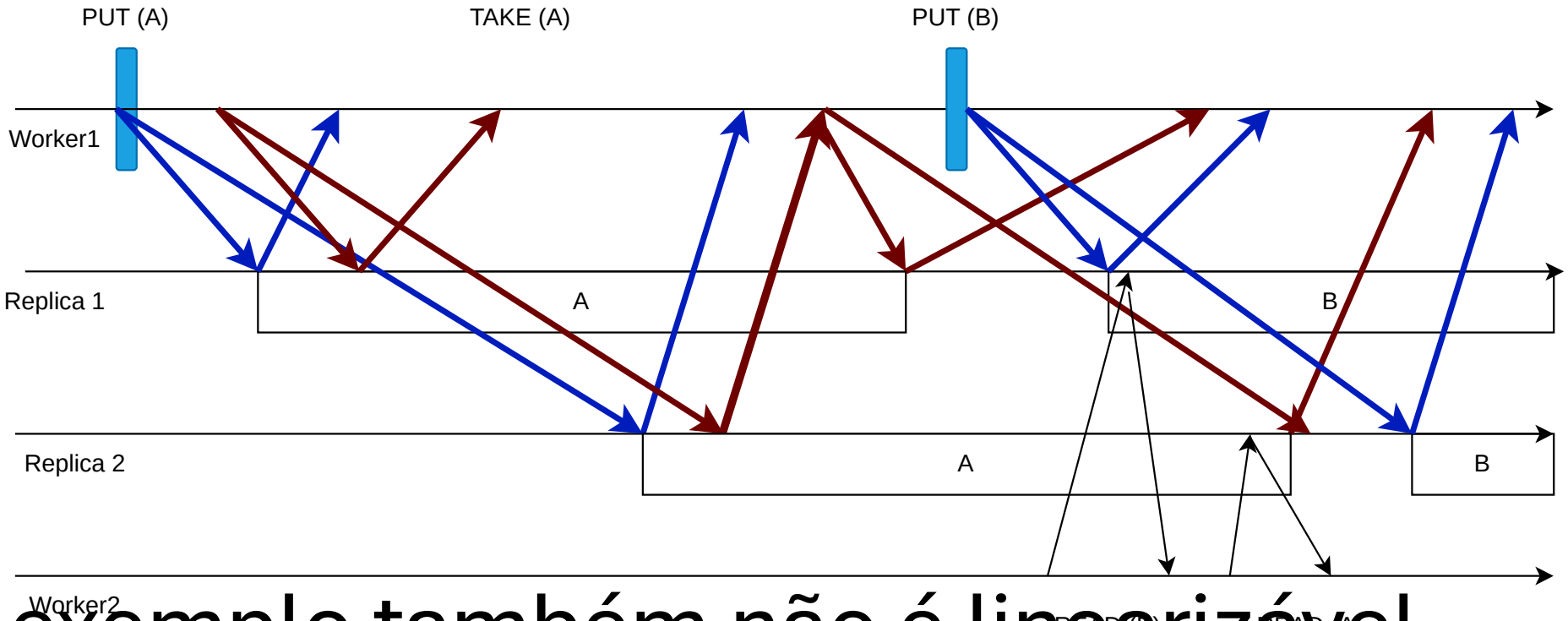
Otimizações para Minimizar a Latência

- O artigo discute duas optimizações importantes:
  - No caso de Put, o cliente pode retornar imediatamente à aplicação, sem esperar pelas respostas
  - No caso do Take, o cliente pode retornar à aplicação assim que souber qual o tuplo que será retirado
    - No caso de take com “wild cards”, isto acontece após a 1ª fase
- Em ambas as otimizações, a operação termina (muito) mais cedo, na perspetiva do cliente
- No entanto, os efeitos das operações só são aplicados nas réplicas mais tarde… Problemas de coerência?

Problemas de coerência? (I)

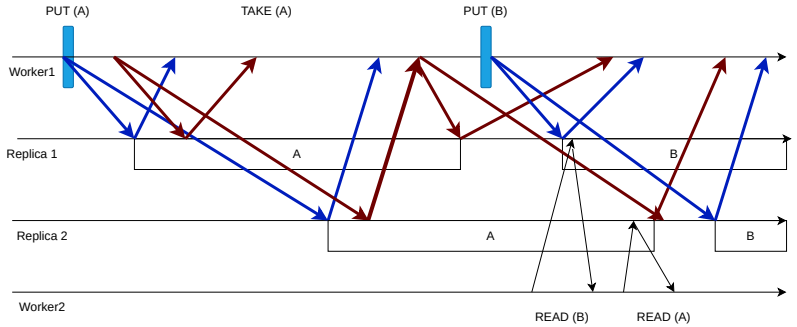
- O mesmo cliente faz “take <x>” e, depois, faz “read <x>”.
  No momento em que o read é invocado, uma réplica pode ainda não ter removido <x> e responde ao cliente com <x>.
- Portanto, o cliente observará uma história não linearizável…
- Como evitar esta anomalia?
  Assegurar que cada réplica executa as operações do mesmo client-id pela ordem de invocação

Problemas de coerência? (II)



Este exemplo também não é linearizável...

Como prevenir a anomalia anterior?



- No exemplo anterior, o read A erróneo após o read B só é possível porque o put se executa na réplica 1 antes do take se concluir na réplica 2.

- Para prevenir esta anomalia:
  - Quando um cliente invoca “put”, esperar até que após o último “take” feito pelo mesmo cliente ter recebido as respostas todas (da fase 2)
  - Só então é que os pedidos de “put” são enviados às réplicas

Bibliografia recomendada

- [Coulouris et al]
  - Secção 6.5.2
    - Parar a partir de “Other approaches”