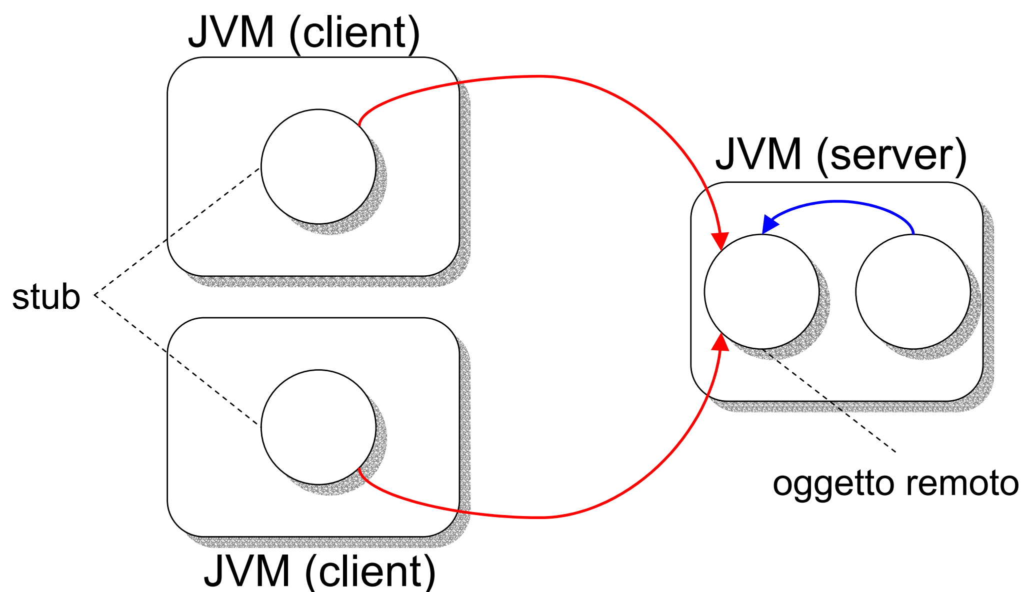
Chiamata remota di metodi

- Garbage Collection Distribuito
- Il meccanismo di “leasing”

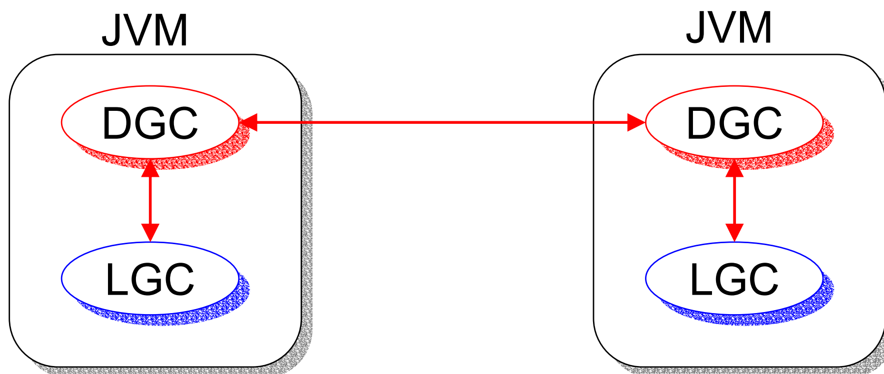
Garbage Collection Distribuito

- Quando non ci sono più riferimenti, **locali** o **remoti**, ad un oggetto, l'oggetto può essere raccolto



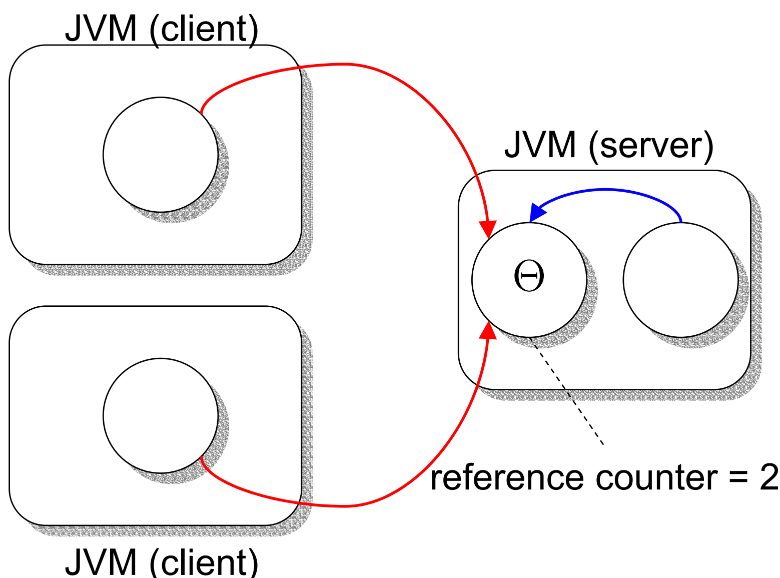
Garbage Collection Distribuito

- In Java, il servizio **Distributed Garbage Collector** (**DGC**) collabora con il servizio Local Garbage Collector (LGC)



Garbage Collection Distribuito

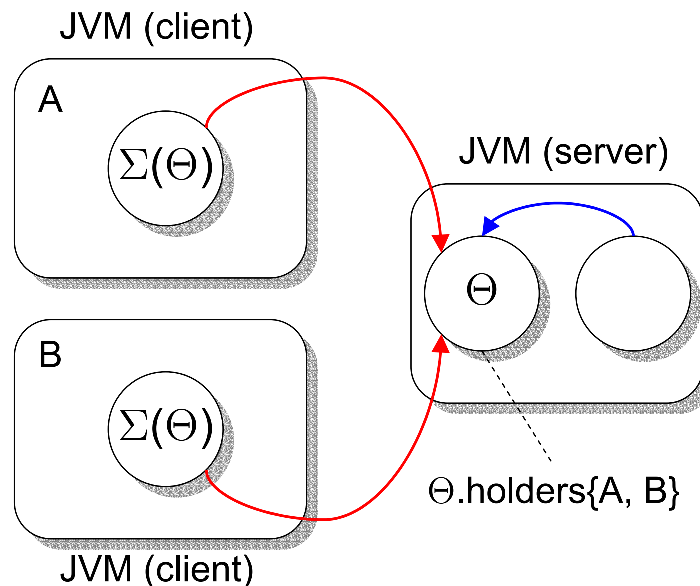
- DGC è basato sul meccanismo del **reference counting**, cioè DGC tiene traccia dei riferimenti a ciascun oggetto remoto



- DGC viene informato di ogni creazione e distruzione di stub in modo da poter incrementare e decrementare, rispettivamente, il reference counter

Garbage Collection Distribuito

- Ogni oggetto remoto Θ è associato all'insieme **holders** che memorizza l'insieme dei clienti che detengono un riferimento all'oggetto Θ

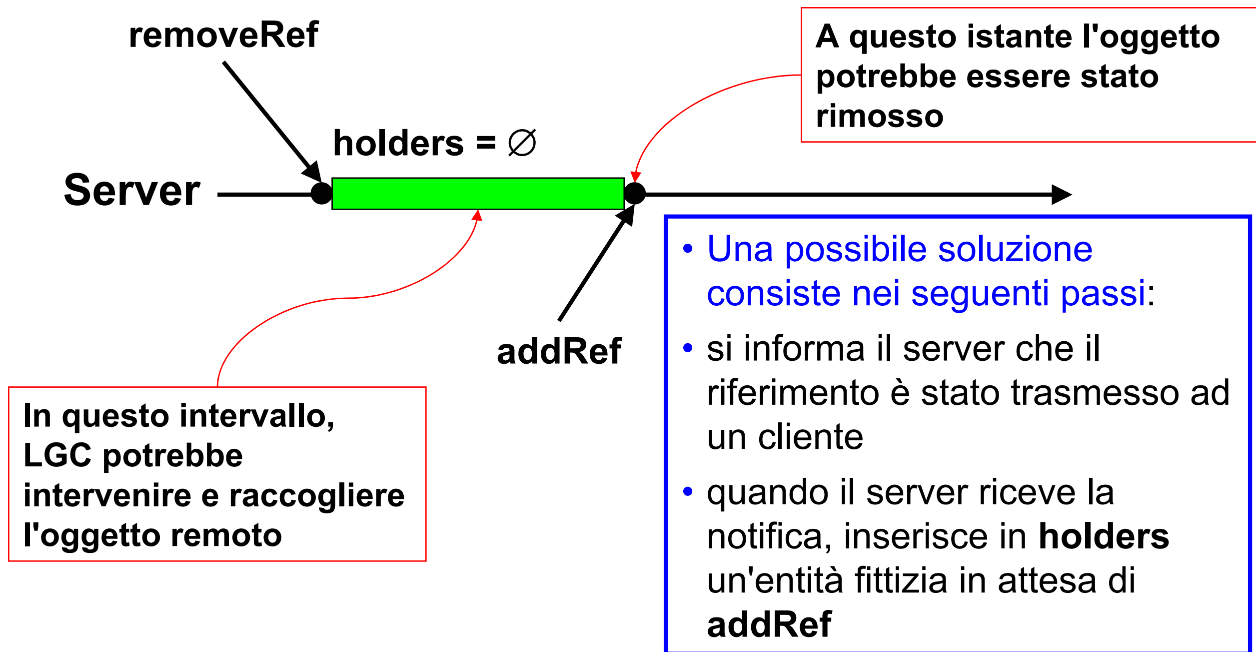


Garbage Collection Distribuito

- **ALGORITMO (semplificato)**
- Quando riceve un riferimento remoto all'oggetto Θ, il cliente C invoca l'operazione remota **addRef**(Θ) sul server e poi crea lo stub Σ(Θ); il server aggiunge C a Θ.holder
- Quando il LGC di C, LGC_C, rileva che lo stub Σ(Θ) può essere raccolto, LGC_C invoca l'operazione **removeRef**(Θ) e cancella lo stub; il server toglie *C* da Θ.holder
- Quando Θ.holder è vuoto e non ci sono riferimenti locali a Θ, l'oggetto Θ può essere reclamato

Garbage Collection Distribuito

- IL PROBLEMA DELLA CORSA CRITICA



- Una possibile soluzione consiste nei seguenti passi:
- si informa il server che il riferimento è stato trasmesso ad un cliente
- quando il server riceve la notifica, inserisce in **holders** un'entità fittizia in attesa di **addRef**

Garbage Collection Distribuito

- Le azioni dell'algoritmo sono "nascoste" nella implementazione dell'oggetto remoto
- Un'applicazione server può ricevere una notifica dell'evento holders = ∅ implementando l'interfaccia **java.rmi.server.Unreferenced**
  - Questa interfaccia ha un solo metodo, **void unreferenced()**, che viene invocato dal RMI runtime

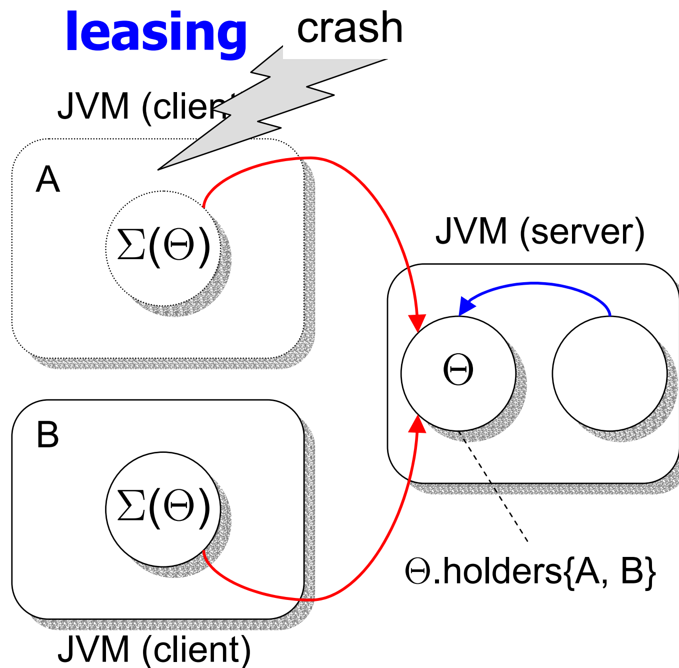
Garbage Collection Distribuito

- Le operazioni **addRef** e **removeRef**
- sono chiamate di procedura remota che seguono la semantica **at-most-once**
- non richiedono una sincronizzazione globale
- non interagiscono con le RMI del livello applicativo perché sono invocate solo quando viene creato o distrutto uno stub

Garbage Collection Distribuito

- **Il servizio DGC di Java tollera i guasti di comunicazione**
- Le operazioni **addRef** e **removeRef** sono idempotenti
- Se l'invocazione di **addRef** ritorna un'eccezione, il cliente non crea lo stub ed invoca l'operazione **removeRef**
- L'operazione **removeRef** ha sempre successo
- Se l'invocazione di **removeRef** dà luogo ad una eccezione interviene il meccanismo di **leasing**

Garbage Collection Distribuito



IL PROBLEMA

- Se il client va in crash non esegue **removeRef**, perciò Θ.holders non diventa mai vuoto e perció l'oggetto Θ non sarà mai reclamato

Il LEASING

- Un riferimento remoto non viene dato per sempre ma affittato per un periodo di tempo massimo prefissato
- Quando il periodo è trascorso, il riferimento viene rimosso
- L'affitto è rinnovabile

Garbage Collection Distribuito

- Il meccanismo di leasing
  - Il server dà un riferimento remoto in affitto (leasing) a ciascun cliente per un tempo massimo prefissato Δ
  - Il periodo di affitto inizia quando il cliente esegue **addRef** e termina quando il cliente esegue **removeRef** e comunque non oltre Δ unità di tempo
  - È responsabilità del cliente rinnovare l'affitto prima che scada
  - A causa di guasti di comunicazione che impediscono al cliente di rinnovare il lease, l'oggetto remoto può "sparire" al cliente
  - Quando l'affitto scade, il server considera il riferimento remoto non più valido e lo rimuove
  - La durata del lease è controllata da **java.rmi.dgc.leaseValue**