Distributed Garbage Collection

- Una delle peculiarità di Java RMI è la presenza di un meccanismo di garbage collection distribuita (DGC) in grado di recuperare la memoria occupata da server remoti inaccessibili, per cui non esistono referenze remote attive.

- Java RMI fornisce tale servizio attraverso un server remoto apposito, unico per ciascuna JVM, con un identificativo ben preciso contenuto nella variabile di sistema `DGC_ID`.

- A differenza di quanto accade con server locali, le referenze remote (cioè gli oggetti stub) si trovano presso i client, su macchine diverse da quella su cui gira il server.

- Di conseguenza la DGC funziona grazie ad una collaborazione tra il runtime RMI della macchina server e quello della macchina client, allo scopo di tenere traccia delle referenze remote presso ciascun client remoto.

- Più precisamente lo stub di un client viene forzato a spedire due messaggi speciali al server.
  - Il primo viene spedito nel momento stesso in cui lo stub è istanziato presso il client, affinchè il server venga a conoscenza che vi è presso il client un nuovo stub attivo.
  - Il secondo messaggio viene spedito dal client al server quando il garbage collector locale del client disalloca la memoria allocata per lo stub. Questo, ovviamente, può accadere solo quando il client non fa più uso dello stub di cui è in possesso.
- Il runtime RMI della macchina server mantiene, per ogni server remoto, un *contatore delle referenze remote attive*.
- Quando lo stub invia il primo messaggio, il runtime RMI presso la macchina server incrementa il contatore del server in questione che viene riconosciuto attraverso il campo `ObjID`.
- Quando poi lo stub invia il secondo messaggio il contatore viene decrementato a segnalare che vi è una referenza remota in meno.

- Quando il contatore di un server remoto arriva a zero allora il runtime RMI della macchina server può attivarsi (vedremo come) per disallocare la memoria occupata dal server remoto attraverso una garbage collection locale presso la macchina server.

Algoritmo di "leasing"

- Il meccanismo appena spiegato, sebbene abbastanza semplice, è presenta tre problemi.

  1. Può accadere che il client, pur avendo uno stub inutilizzabile, si attardi ad invocare il proprio garbage collector locale per disallocarlo. Magari perchè sta eseguendo un qualche task ad alta priorità e non ha esigenza di liberare memoria. In tal caso il client impedisce al runtime RMI della macchina server di fare la DGC del server remoto.

  2. La macchina client potrebbe crashare. In tal caso il client potrebbe non aver spedito al runtime RMI della macchina

server il secondo messaggio per consentire il decrement del contatore.

3. Infine possono sorgere problemi di rete che impediscano al server di ricevere i due messaggi inviati dallo stub del client.

- Il primo di questi tre problemi è ovviamente senza soluzione poichè la specifica di Java è molto chiara nell'indicare che non c'è modo di forzare l'esecuzione di un garbage collector (locale).

- I restanti due problemi sono invece superati da Java RMI rendendo le referenze remote *temporanee*, attraverso un meccanismo chiamato di *leasing*.

- L'algoritmo di leasing funziona nel seguente modo:

  1. Lo stub presso il client invoca il server per chiedere un "lease" per un periodo di tempo. Di default un "lease" dura 10 minuti, ma lo si può configurare usando la proprietà `java.rmi.dgc.leasevalue`.

2. Il server risponde concedendo il “lease” per un certo tempo, che non necessariamente coincide con quanto chiesto dal client.
3. Durante questo periodo il runtime RMI della macchina server sa che c’è uno stub attivo presso il client ed incrementa il contatore del server remoto.
4. Quando il “lease” scade, se il client non ha chiesto un’estensione, il contatore viene automaticamente decrementato.
5. I clients automaticamente cercano di rinnovare i “lease” associato allo stub fintantochè quest’ultimo non è stato garbage collected.

- Si noti che grazie all’algortimo di leasing, se un client va in crash oppure la rete va giù, allora l’applicazione client non è in grado di rinnovare il “lease” del proprio stub e di conseguenza il runtime RMI della macchina server decrementa il contatore relativo al server remoto.

Il server DGC

- Come già detto il "distributed garbage collector" è esso stesso un server RMI che implementa un'interfaccia remota `DGC` definita nel package `java.rmi.dgc`.

- Questa interfaccia ha solo due metodi:

```
public void  clean(ObjID[] ids, long seqNum, ....)
public Lease dirty(ObjID[] ids, long seqNum, Lease lease)
```

- Entrambi i metodi sono invocati automaticamete dal runtime RMI del client.

- `clean()` è invocato dal client quando non utilizza più uno stub. Tecnicamente, grazie ai "lease" si potrebbe pure evitare di invocare tale metodo poichè la referenza dopo un po' scadrebbe. Ma per motivi di efficienza è comunque importante invocarlo appena possibile informando il server che lo stub non è più attivo.

- Il metodo `dirty()` è invece invocato dal client per richiedere un "lease". Tra i parametri viene per l'appunto passato il"lease" richiesto dal client che non necessariamente coincide con quello accordato dal server.

L'interfaccia `java.rmi.server.Unreferenced`

- L'interfaccia `Unreferenced` fornisce un meccanismo per la gestione della garbage collection di server remoti.

- Un server remoto non è obbligato ad implementare l'interfaccia `Unreferenced`, ma se lo fa deve fornire un'implementazione al metodo `Unreferenced.unreferenced`.

- Il metodo `unreferenced` è invocato automaticamente dal runtime RMI del server quando il sottosistema DGC segnala che non vi sono più client remoti. Ovvero, quando il contatore associato alle referenze remote del server giunge a zero.

- Si ricordi che in generale il registro RMI presso l'host del server potrebbe contenere uno stub al server remoto. In questo caso si tratterebbe di uno *stub locale* (perchè sullo stesso host), ovvero una *referenza locale al server remoto.*

- L'interfaccia `Unreferenced` è tipicamente usata per due scopi:
  1. Per consentire certe attività del server in assenza di clients. Ovvero riorganizzazione dei dati, garbage collection, ect.
  2. Consentire al server di terminare senza danneggiare eventuali clients.
     - Ricordiamo che un server remoto esportato è automaticamente de-esportato quando viene rimosso dal garbage collector locale.
     - Un server remoto può essere sottoposto al garbage collector locale solo quando non esistono sue referenze né remote e né locali.
     - Perciò se si vuole che il server sia de-esportato e sottoposto a garbage collection locale, bisognerà rimuovere le reference locali restanti.

– Solo dopo aver rimosso sia le referenze remote che le referenze locali, è possibile invocare il garbage collector (`System.gc()`) per rimuovere la memoria allocata per il server remoto.
– Tutte queste operazioni devono essere eseguite all'interno del metodo `unreferenced` che il programmatore deve avere cura di implementare.