Lezione n.9
LPR-B-08
RMI CallBacks - Miscellanea
3/12/2008

Andrea Corradini
Laura Ricci

Sommario

- RMI: il meccanismo delle callbacks

- Thread Miscellanea:
  - Thread Pool: politiche di saturazione
  - Blocchi Sincronizzati

RMI: PASSAGGIO DI PARAMETRI

- Nell'invocazione di un metodo con RMI, i parametri vengono passati nel seguente modo:
  - parametri di tipo primitivo vengono passati per valore
  - parametri di tipo riferimento vengono serializzati e il server ne crea una copia
    - Qundi una modifica fatta dal server non è visibile dal client
  - parametri di tipo "remoto" vengono passati come riferimenti

IL MECCANISMO DELLE CALLBACK: MOTIVAZIONI

Meccanismo RMI prevede:

- comunicazione **unidirezionale** (dal client al server)
- comunicazione **sincrona**, rendez-vouz esteso: il client invoca un metodo remoto e si blocca finchè il metodo non termina

Ma in molte applicazioni

- il client è interessato ad un evento che si verifica sul server e notifica il suo interesse al server (per esempio utilizzando RMI)
- il server **registra** che il client è interessato in quell'evento
- quando l'evento si verifica, **il server notifica** ai clients interessati l'accadimento dell'event

IL MECCANISMO DELLE CALLBACK: MOTIVAZIONI

Esempi di applicazioni:

- Un utente partecipa a un gruppo di discussione (es: Facebook) e vuol essere avvertito quando un nuovo utente entra nel gruppo.
- Lo stato di un gioco multiplayer viene gestito da un server. I giocatori notificano al server le modifiche allo stato del gioco. Ogni giocatore deve essere avvertito quando lo stato del gioco subisce delle modifiche.
- Gestione distribuita di un'asta: un insieme di utenti partecipa ad un'asta distribuita. Ogni volta che un utente fa una nuova offerta, tutti i partecipanti all'asta devono essere avvertiti.

IL MECCANISMO DELLE CALLBACK: MOTIVAZIONI

- Come può essere avvisato il client che un evento si è verificato sul server?

  - **Polling**: il client interroga ripetutamente il server, per sapere se si è verificato l'evento atteso. L'interrogazione può avviene tramite mediante **RMI**
    - Svantaggio: inefficiente, spreco di risorse del sistema

  - **Registrazione** dei clients interessati agli eventi e successiva notifica (asincrona) del verificarsi dell'evento al client da parte del server
    - Problema: quale meccanismo può usare il server per avvisare il client?

RMI: IL MECCANISMO DELLE CALLBACK

- Il **meccanismo delle callback** permette di utilizzare **RMI** sia per l'invocazione client-server (registrazione del client) che per quella server-client (notifica del verificarsi di un evento).
- Come funziona?
  - Il server definisce un'interfaccia remota **ServerInterface** con un **metodo remoto** che serve al client per **registrarsi**
  - Il client definisce un'interfaccia remota **ClientInterface** che definisce un **metodo remoto** usato dal server per **notificare** un evento al client
  - Il client conosce la **ServerInterface** e ottiene il puntatore all'**oggetto remoto** tramite il registry

RMI: IL MECCANISMO DELLE CALLBACK

- Il client invocando il **metodo remoto** per **registrarsi** passa al server un riferimento **RC** a un oggetto che implementa la **ClientInterface**
- Il server memorizza **RC** in una sua struttura dati (ad esempio, un Vector)
- Quando deve notificare, il server utilizza **RC** per invocare il **metodo remoto** di notifica definito dal client.
- In questo modo si rende 'simmetrico' il meccanismo di RMI, ma…
  - il client **non registra** l'oggetto remoto in un rmiregistry, ma passa un riferimento a tale oggetto al server, al momento della registrazione

CHE SIGNIFICA "CALLBACK"?

Da Wikipedia: "In programmazione una callback è una **funzione specializzata** che viene passata come **parametro** a **un'altra funzione** (che invece è **generica**). Questo permette alla funzione generica di compiere un lavoro specifico attraverso la callback."

- **Esempio**: la funzione generica **quicksort** prende come argomento l'array da ordinare e una funzione **callback** per confrontare gli elementi.
- Nel contesto Object Oriented, una **callback** è un'istanza che fornisce un'implementazione di un metodo specificato in un'interfaccia.
- **Esempio nelle API Java**: Uso di **Comparator** nei metodi di **sorting** della classe **Arrays**

CALLBACKS: UN ESEMPIO

Lato Server:

- Interfaccia remota che definisce metodi per: (1) Contattare il server: metodo SayHello( ) (2) **Registrare/cancellare** una **callback:**
- Implementazione dell'interfaccia
- Esportazione e pubblicazione su registry di oggetto remoto

Lato Client:

- Intefaccia remota che definisce metodo remoto (callback)
- Implementazione di interfaccia remota
- Programma principale che:
  - Registra una callback presso il server: sarà usata per notificare al client i contatti stabiliti da clients con il metodo SayHello( ).
  - Effettua un numero casuale di richieste del metodo SayHello( )
  - Cancella la propria registrazione

L'INTERFACCIA REMOTA DEL SERVER

```
import java.rmi.*;
    public interface CbServerInt extends Remote {

    /* metodo di notifica */
    public String sayHello(String name) throws RemoteException;

    /* registrazione per il callback */
    public void registerForCallback(CbClientInt callbackClient)
                    throws RemoteException;
    /* cancella registrazione per il callback */
    public void unregisterForCallback(CbClientInt callbackClient)
                    throws RemoteException;
}
```

L'IMPLEMENTAZIONE DEL SERVER

```
import java.rmi.*; import java.rmi.server.*;import java.util.*;
public class CbServerImpl extends UnicastRemoteObject
                                          implements CbServerInt{
    /* lista dei client registrati */
    private List<CbClientInt> clients;

    /* crea un nuovo servente */
    public CbServerImpl( ) throws RemoteException {
        clients = new ArrayList<CbClientInt>( );
    }
    /* continua */
```

L'IMPLEMENTAZIONE DEL SERVER

```
public synchronized void registerForCallback(CbClientInt callbackClient)
                                        throws RemoteException{
    if (!clients.contains(callbackClient)) {
        clients.add(callbackClient);  }
    System.out.println(" New client registered." ); }
/* annulla registrazione per il callback */
public synchronized void unregisterForCallback(CbClientInt callbackClient)
                                        throws RemoteException{
    if (clients.remove(callbackClient)) {
        System.out.println("Client unregistered");
    }else{
        System.out.println("Unable to unregister client.");
    }}
```

L'IMPLEMENTAZIONE DEL SERVER

```
/* metodo di notifica
 * quando viene richiamato, fa il callback a tutti i client registrati */
public String sayHello (String name) throws RemoteException {
      doCallbacks(name);
      return "Hello, " + name + "!";   }
private synchronized void doCallbacks(String name) throws RemoteException{
      System.out.println("Starting callbacks.");
      Iterator<CbClientInt> i = clients.iterator( );
      int numeroClienti = clients.size( );
      while (i.hasNext()) {
            CbClientInt client = i.next();
            client.notifyMe(name); }
      System.out.println("Callbacks complete."); } }
```

L'ATTIVAZIONE DEL SERVER

```
import java.rmi.server.*; import java.rmi.registry.*;
public class CbServer   {
      public static void main(String[  ] args) {
            try { /* registrazione presso il registry */
                  System.out.println("Binding CallbackHello");
                  CbServerImpl server =  new CbServerImpl(    );
                  String name = "CallbackHelloServer";
                  Registry registry = LocateRegistry.getRegistry ("localhost",2048);
                  registry.rebind  (name, server);
                  System.out.println("CallbackHello bound");
            } catch (Exception e) {
                  e.printStackTrace();
                  System.exit(-1); } } }
```

L'INTERFACCIA DEL CLIENT

```
import java.rmi.*;
public interface CbClientInt extends Remote {
    /*  Metodo invocato dal  server per effettuare
        una callback a un client remoto. */
public void notifyMe(String message) throws RemoteException;
}
```

notifyMe(...)

è il metodo esportato dal client e che viene utilizzato dal server per la notifica di un nuovo contatto da parte di un qualsiasi client. Viene notificato il nome del client che ha contattato il server.

L'IMPLEMENTAZIONE DEL CLIENT

```
import java.rmi.*; import java.rmi.server.*;


public class CbClientImpl  extends UnicastRemoteObject
                                        implements CbClientInt {
/* crea un nuovo callback client */
     public CbClientImpl( ) throws RemoteException {
          super( ); }

/* metodo che può essere richiamato dal servente */
     public void notifyMe(String message) throws RemoteException {
          String returnMessage = "Call back received: " + message;
          System.out.println( returnMessage);  } }
```

ATTIVAZIONE DEL CLIENT

```
import java.rmi.*; import java.rmi.server.*; import java.rmi.registry.*;
public class CbClient {
    public static void main(String args[ ]) {
    try {System.out.println("Cerco CallbackHelloServer");
        Registry registry = LocateRegistry.getRegistry("localhost", 2048);
        String name = "CallbackHelloServer";
        /* crea stub di oggetto remoto */
        CbServerInt h =(CbServerInt) registry.lookup(name);
        /* si registra per il callback */
        System.out.println("Registering for callback");
        CbClientImpl callbackObj = new CbClientImpl( );
        h.registerForCallback(callbackObj);
```

ATTIVAZIONE DEL CLIENT

```
        /* accesso al server - fa una serie casuale di 5-15 richieste */
        int n = (int) (Math.random( ) * 10 + 5);
        String nickname=  "mynick";
        for (int i=0; i< n; i++) {
              String message = h.sayHello(nickname);
              System.out.println( message);
              Thread.sleep(1500);  }
        /* cancella la registrazione per il callback */
        System.out.println("Unregistering for callback");
        h.unregisterForCallback(callbackObj);
} catch (Exception e) {
System.err.println("HelloClient exception: " +e.getMessage(  ));
e.printStackTrace(); System.exit(-1); }}}
```

RMI:ECCEZIONI

- Eccezione che viene sollevata se non trova un servizio di registry su quella porta. Esempio:

  HelloClient exception: Connection refused to host: 192.168.2.103; nested exception is: java.net.ConnectException: Connection refused: connect

- Eccezione sollevata se si tenta di registrare più volte lo stesso stub con lo stesso nome nello stesso registry

  Esempio

  CallbackHelloServer exception: java.rmi.AlreadyBoundException: CallbackHelloServer java.rmi.AlreadyBoundException: CallbackHelloServer

ESERCIZIO 1: ELEZIONI CON CALLBACK

La scorsa lezione precedente è stato assegnato un esercizio per la gestione elettronica di una elezione a cui partecipano un numero prefissato di candidati. Si chiedeva di realizzare un server RMI che consentisse al client di votare un candidato e di richiedere il numero di voti ottenuti dai candidati fino ad un certo punto.

Modificare l'esercizio in modo che il server notifichi ogni nuovo voto ricevuto a tutti i clients che hanno votato fino a quel momento. La registrazione dei clients sul server avviene nel momento del voto.

ESERCIZIO 2: GESTIONE FORUM

Si vuole implementare un sistema che implementi un servizio per la gestione di forum in rete. Un forum è caratterizzato da un argomento su cui diversi utenti, iscritti al forum, possono scambiarsi opinioni via rete.

Il sistema deve prevedere un server RMI che fornisca le seguenti funzionalità:

a) apertura di un nuovo forum, di cui è specificato l'argomento (esempio: giardinaggio)

b) registrazione ad un forum, di cui è specificato l'argomento

c) inserimento di un nuovo messaggio indirizzato ad un forum identificato dall'argomento (es: è tempo di piantare le viole, indirizzato al forum giardinaggio); il messaggio deve essere inviato agli utenti iscritti al forum

d) reperimento dell'ultimo messaggio inviato ad un forum di cui è specificato l'argomento.

Quindin il messaggio può essere richiesto esplicitamente dal client oppure può essere notificato ad un client precedentemente registrato.

SERVER PROGRAMMING

- **Server Web, Mail Server, File Server, Database Server** sono tutti caratterizzati dalla seguente struttura:

```
ServerSocket socket = new ServerSocket(80); // HTTP
while (true){
    Socket connection = socket.accept( );
    handleRequest(connection); }
```

- La gestione di ogni servizio (**handleRequest**) è indipendente dalla gestione degli altri
- Esempio: la gestione di un messaggio inviato a un mail server non dipende da quella degli altri messaggi elaborati contemporaneamente
- E' naturale associare un thread diverso a ogni richiesta di servizio.

SERVER PROGRAMMING

```
import java.net.*; import java.util.concurrent.*; import java.io.*;

public class ServerPool {

private static final int NTHREADS = 100;

private static final Executor exec=Executors.newFixedThreadPool(NTHREADS);

    public static void main(String [ ] args) throws IOException{

    ServerSocket socket = new ServerSocket(80);

    while (true) {

        final Socket connection = socket.accept();

        Runnable task = new Runnable( ){  // classe anonima
            public void run( )   {  handleRequest(connection); }  };

        exec.execute(task);  }   }   }
```

THREAD POOL: POLITICHE DI SATURAZIONE

- Politica di saturazione:
  - serve se si usano thread pool con code di dimensione limitata
  - indica cosa si deve fare se il pool è saturo, cioè la coda è piena ed i threads del pool sono tutti attivi
- Strategia di default: **abort**, il task viene rifiutato e viene sollevata una **RejectedExecutionException**
- E' possibile definire una politica *ad hoc* mediante un **Rejected Execution Handler**
- JAVA mette a disposizione diversi **Rejected Execution Handler**, ognuno dei quali implementa una diversa politica di saturazione
- Selezione della politica: setRejectedExecutionHandler

THREAD POOL:POLITICHE DI SATURAZIONE

Politiche di saturazione: quando viene sottomesso un task T e la coda è piena, possono essere adottate le seguenti politiche

- **abort**: T viene scartato e viene sollevata un'eccezione
- **discard policy**: rifiuta T, ma non solleva alcun tipo di eccezione
- **discard oldest** scarta il primo task della coda (quello che avrebbe dovuto essere eseguito successivamente) e inserisce T in coda
- **caller-runs**
  - non scarta il task e non solleva eccezioni.
  - cerca di rallentare (throttling, to throttle = strozzare) il flusso dei tasks restituendo alcuni task al chiamante per l'esecuzione. Il task non viene eseguito in un thread del pool, ma nel thread che ha invocato la execute

THREAD POOL: POLITICHE DI SATURAZIONE

```
import java.util.concurrent.*;

public class prova {

public static void main (String args[ ]) {
      ThreadPoolExecutor executor =
                  new ThreadPoolExecutor(10,  // corePoolSize
                                         11,  // maxPoolSize
                                         0L,  // keepAliveTime
                                TimeUnit.MILLISECONDS,  // timeUnit
                               new LinkedBlockingQueue<Runnable>(100));
      executor.setRejectedExecutionHandler  (
                                new ThreadPoolExecutor.CallerRunsPolicy());
      }
}
```

THREAD POOL:POLITICHE DI SATURAZIONE

- Consideriamo un **Web Server**
  - riceve richieste di servizio da parte dei clients
  - ogni richiesta corrisponde a un diverso task
- Supponiamo che il Web Server usi un thread pool con una coda a dimensione limitata e una politica di saturazione caller-runs.
- Se tutti i threads sono occupati e la coda è piena, la successiva richiesta di servizio viene eseguita nel thread che ha invocato la execute (il main thread del server)
- Poiché l'esecuzione del servizio richiesto richiederà un intervallo di tempo ΔT, il main thread non sottometterà ulteriori taks per l'esecuzione al pool, durante ΔT
- Alcuni thread del pool possono terminare l'esecuzione del task assegnato durante ΔT, rendendosi disponibili a nuove sottomissioni

THREAD POOL: POLITICHE DI SATURAZIONE

- Inoltre, il main thread, impiegato nell'esecuzione del task rifiutato dal pool, non accetta ulteriori connessioni (non esegue l'accept sul ServerSocket) durante l'intervallo di tempo ΔT
- Le richieste di servizio vengono quindi accodate a livello TCP
- Se l'overload persiste, sarà il livello TCP che eventualmente inizierà a scartare richieste, quando la sua coda sarà, a sua volta, satura
- Conclusione: se il server è sovraccarico, si può implementare una graceful degradation, spostando gradualmente il sovraccarico
  - dai thread del pool alla coda associata al pool,
  - dalla coda all'applicazione,
  - dall'applicazione al livello TCP
  - Il livello TCP è in grado di bloccare il client, mediante il meccanismo di controllo del flusso

BLOCCHI SINCRONIZZATI

- Sincronizzazione di un blocco di codice

```
synchronized (obj)
    { // blocco di codice
    }
```

- L'oggetto obj può essere quello su cui è stato invocato il metodo che contiene il codice (this) oppure un altro oggetto
- Il thread che esegue il blocco sincronizzato deve acquisire la lock sull'oggetto obj
- La lock viene rilasciata nel momento in cui il thread termina l'esecuzione del blocco (es: return, throw, esecuzione dell'ultima istruzione del blocco)
- La lock non viene rilasciata nel caso in cui si invochi un altro metodo all'interno del blocco di codice (lock rientrante)

BLOCCHI SINCRONIZZATI

- Utilizzo:
  - ridurre la lunghezza di una sezione critica
  - rendere indivisibile un frammento di codice che invoca due o più metodi synchronized
- L'esempio presentato nei lucidi successivi mostra in modo operativo

  il vantaggio di diminuire la lunghezza di una sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
double A,B.....
public void setValues (int x, double r){
    double tempA= // questa elaborazione richiede molto tempo...
    double tempB= // questa elaborazione richiede molto tempo...
    synchronized ( this ) {
        A = tempA;
        B = tempB; } }
```

- L'elaborazione che richiede molto tempo viene eseguita fuori dalla sezione critica ed i risultati vengono assegnati a variabili locali
- la sezione critica contiene solo l'aggiornamento delle variabili di istanza del metodo
- si riduce il tempo in cui un thread che ha acquisito la lock rimane all'interno della sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

La classe **Pair** incapsula

- una coppia di valori interi (x,y)
- un valore intero accesscount.

La coppia di valori deve essere aggiornata in modo atomico e in mutua esclusione rispetto all'aggiornamento della variabile **accesscount**

```
public class Pair {
    private int x, y;
    private int accesscount = 0;
    public Pair (int x, int y)
        {  this.x = x;   this.y = y;  };
    public synchronized void accessincrement( ) { accesscount++;};
    public synchronized int accessreturn( ){ return accesscount;};
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

Per l'aggiornamento della coppia di valori, si definiscono due diversi metodi

- **increment_syn_method**( ): utilizza un metodo sincronizzato
- **increment_syn_block**( ): utilizza un blocco sincronizzato
- **Nota Bene**: la sleep simula un'operazione computazionalmente costosa

```
public synchronized void increment_syn_method(  ){
      x++;  y++;
      try {Thread.sleep(100);} catch(Exception e)  {  };  }
public void increment_syn_block( ){
      synchronized(this){
            x++;  y++;  }
      try {Thread.sleep(100);} catch(Exception e)  {  };  }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class Thread_Syn_Method  extends Thread {
      Pair p;
      public Thread_Syn_Method(Pair p) {this.p=p;};
      public void run(   ) {
            while (true)  {
                  p.increment_syn_method(  );   }  }  }
public class Thread_Syn_Block extends Thread{
      Pair p;
      public Thread_Syn_Block(Pair p) {this.p=p;};
      public void run(   ){
            while (true)  {
                  p.increment_syn_block(); }   }    }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class CountAccess extends Thread {

        Pair p;

        public CountAccess(Pair p){

        this.p = p;

        }

        public void run(  ) {
              while (true) p.accessincrement( );
        }
  }
```

OTTIMIZZAZIONE SEZIONI CRITICHE

```
public class MainPair {
public static void main(String args[]){
      Pair syn_method_pair = new Pair(1,2);
      Pair syn_block_pair = new Pair(3,4);
      new Thread_Syn_Method(syn_method_pair).start( );
      new CountAccess(syn_method_pair).start( );
      new Thread_Syn_Block(syn_block_pair).start( );
      new CountAccess(syn_block_pair).start( );
      try      {Thread.sleep(200);}      catch(Exception e)  {  }
      System.out.println("Accessi con blocco sincronizzato " +
            syn_block_pair.accessreturn( ) +
            "\nAccessi con metodo sincronizzato " +
            syn_method_pair.accessreturn( ));  }}
```

OTTIMIZZAZIONE SEZIONI CRITICHE

Risultati di alcune esecuzioni:

Accessi con blocco sincronizzato 2258106 Accessi con metodo sincronizzato 2843

Accessi con blocco sincronizzato 2242580 Accessi con metodosincronizzato 2

Accessi con blocco sincronizzato 3749759 Accessi con metodo sincronizzato 1252

Conclusioni:

- il thread che accede all'oggetto coppia con **blocco** sincronizzato (invece che con un metodo sincronizzato) consente un maggior numero di accessi al thread **CountAccess**, che viene eseguito concorrentemente
- La versione che utilizza il **blocco** sincronizzato aumenta il numero di accessi concorrenti effettuati sull'oggetto coppia
- NOTA BENE: La differenza nel numero di accessi dipende ovviamente dal fatto che il programma simula la computazione costosa con una sleep( ). Se si sincronizza tutto il metodo, quando il thread esegue la sleep rilascia il processore, ma non la lock().