Laboratorio di Reti
Lezione 11
RMI:
Multithreading

07/12/2021

Federica Paganelli

Source: slides corso LPR A AA 2017/18

RMI E CONCORRENZA

- Analizziamo le caratteristiche di un servizio remoto
  - non analizzeremo dettagliatamente l'implementazione
  - studiamo il comportamento tramite un insieme di esempi

- Vogliamo capire:
  - poiché esiste un solo oggetto remoto, i metodi di quell'oggetto possono essere invocati in modo concorrente da client diversi o da thread diversi dello stesso client?
  - in caso affermativo, viene creato un thread per ogni richiesta? Per ogni client?
  - cosa accade se non sincronizzo opportunamente gli accessi sull'oggetto remoto?

RMI E CONCORRENZA

Dalla documentazione ufficiale:

"A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping invocations to threads. Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe"

L'oggetto può essere chiamato da thread diversi, gestiti da RMI

- Invocazioni di metodi remoti provenienti da client diversi (diverse JVM) sono tipicamente eseguite da thread diversi
  - consente di non bloccare un client in attesa della terminazione dell'esecuzione di un metodo invocato da un altro client
  - ottimizza la performance del servizio remoto
- Invocazioni concorrenti provenienti dallo stesso client (ad esempio se le chiamate si trovano in due thread diversi del client) possono essere eseguite dallo stesso thread o da thread diversi.

RMI E CONCORRENZA

- La politica di JAVA RMI di implementare automaticamente multithreading di chiamate diverse presenta il vantaggio di evitare all'utente di scrivere codice per i thread (server side)

- il server non risulta thread safe:
- richieste concorrenti di client diversi possono portare la risorsa ad uno stato inconsistente

- L'utente che sviluppa il server deve assicurare che l'accesso all'oggetto remoto sia correttamente sincronizzato (metodi synchronized, locks....)

RMI E CONCORRENZA: UNDER THE HOOD

Per verificare se i metodi dell'oggetto remoto sono invocati in modo concorrente,

- definiamo un oggetto remoto che esporta due metodi
- ogni metodo non fa altro che stampare per un certo numero di volte che è in esecuzione
- attiviamo due client: uno invoca il primo metodo, uno il secondo
  - si ottiene un interleaving delle stampe?

```
public interface ThreadsInt  extends java.rmi.Remote {
public void methodOne()
                    throws java.rmi.RemoteException;
public void methodTwo()
                    throws java.rmi.RemoteException;
}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
import java.rmi.*;
public class Threadsimpl extends RemoteObject implements
ThreadsInt {
  public Threadsimpl() throws RemoteException
    {super();}
  public void methodOne() throws RemoteException {
    long TimeOne = System.currentTimeMillis();
    for(int index=0;index<25;index++)
      { System.out.println("Method ONE executing");
        // Inserito un ritardo di circa mezzo secondo
        do{
           } while ((TimeOne+500)>System.currentTimeMillis());
       TimeOne = System.currentTimeMillis();
       }
 }
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public void methodTwo() throws RemoteException {
     long TimeTwo = System.currentTimeMillis();
     for(int index=0;index<25;index++)
          {
           System.out.println("Method TWO executing");
           // Inserito un ritardo di circa mezzo secondo
           do{
             }while ((TimeTwo+500)>System.currentTimeMillis());
           TimeTwo = System.currentTimeMillis();
             }
   }
}
```

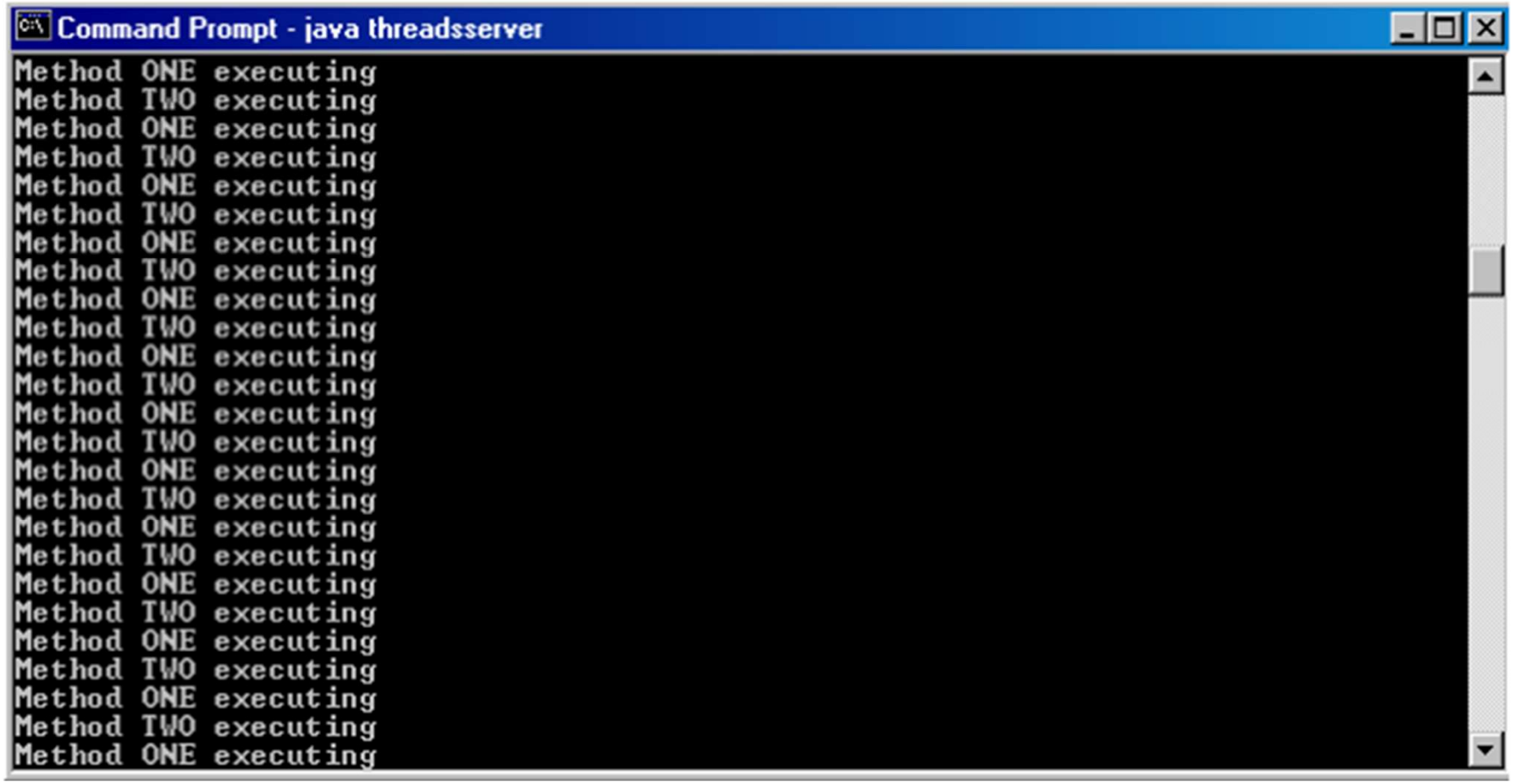
RMI E CONCORRENZA: UNDER THE HOOD

```
public class Threadserver {
   public Threadserver(int porta) {
     try {LocateRegistry.createRegistry(porta);
        Registry r=LocateRegistry.getRegistry(porta);
        System.out.println("Registro Reperito");
        Threadsimpl c = new Threadsimpl();
        ThreadsInt stub =(ThreadsInt)
                    UnicastRemoteObject.exportObject(c, 0);
        r.rebind("Threads", stub); }
     catch (Exception e) {
                  System.out.println("Server Error: " + e); }  }
    public static void main(String args[]) {
                                  new threadserver(args[0]);  }}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public class ThreadsClient {
    public static void main(String[] args) {
    try {
         Registry r= LocateRegistry.getRegistry(args[0]);
         ThreadsInt c = (ThreadsInt) r.lookup("Threads");
         if (args[1].equals("one"))
                  c.methodOne();
         else if (args[1].equals("two"))
                 c.methodTwo();
                 else System.out.println("Error: correct usage -
treadsclient port {one|two}");            }  catch (Exception
e){  }}}
```

RMI E CONCORRENZA: UNDER THE HOOD



Una possibile traccia di esecuzione ottenuta attivando un client con argomento "one" ed uno con argomento "two"

RMI E CONCORRENZA: UNDER THE HOOD

- l'esecuzione del metodo TWO è iniziata prima che l'esecuzione del metodo ONE sia terminata
  - ciò implica che ai due client sono stati associati due diversi threads, che invocano i metodi dell'oggetto remoto in modo concorrente

- se si vuole rendere “atomica” l'esecuzione di un metodo occorre utilizzare meccanismi opportuni di sincronizzazione (ad es. metodi synchronized)

- diversi modelli di esecuzione delle richieste provenienti dai client per l'esecuzione di metodi dell'oggetto remoto
  - prelevare le richieste da una coda e servirle sequenzialmente
  - un thread per ogni richiesta. Il thread invoca i metodi dell'oggetto remoto

RMI E CONCORRENZA: UNDER THE HOOD

- Come vengono trattate le richieste provenienti da uno stesso client?
- Se il client è sequenziale, ci può essere al massimo una richiesta pendente o in esecuzione per volta.
- Se il client attiva più threads, questi possono eseguire in parallelo richieste di esecuzione di metodi sull'oggetto remoto
- Le richieste vengono eseguite in sequenza o in maniera concorrente?
- Il programma successivo indaga questo aspetto (si riferisce al servizio remoto definito nei lucidi precedenti).

RMI E CONCORRENZA: UNDER THE HOOD

```
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.lang.*;
public class ThreadsClientMod {
     public static void main(String[] args) {
          try {
               Registry r= LocateRegistry.getRegistry(2800);
               ThreadsInt c = (ThreadsInt) r.lookup("Threads");
               OneThread t1 = new OneThread(c);
               t1.start();
               TwoThread t2 = new TwoThread(c);
               t2.start();
            }   catch (Exception e){}}}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public class OneThread extends Thread{
    ThreadsInt x;
    public OneThread(ThreadsInt c) {
      this.x=c;
    }
   public void run() {
      try {
          x.methodOne();}
       catch(Exception e){}
   }
}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public class TwoThread extends Thread{
     ThreadsInt x;
     public TwoThread(ThreadsInt c) {
       this.x=c;
     }
     public void run() {
      try {
          x.methodTwo();}
       catch(Exception e){}
     }
}
```

- l'esecuzione di questo programma consente di capire se invocazioni concorrenti da parte dello stesso client vengono eseguite in modo concorrente o meno, sulla propria macchina

RMI E THREAD SAFETY: RACE CONDITIONS

```
import java.rmi.*;
    public interface RMIThreadServer extends Remote {
    public void update() throws RemoteException;
    public int read() throws RemoteException;
     }
```

RMI E THREAD SAFETY: RACE CONDITIONS

```
import java.rmi.*; import java.rmi.server.*; import java.rmi.registry.*;
public class RMIThreadServerImpl extends UnicastRemoteObject implements
      RMIThreadServer{
  private volatile int counter=0;
  private final int MAXCOUNT = 9000;
  public RMIThreadServerImpl() throws RemoteException  {super(); }
  public void update() {
      int i;
      Thread p = Thread.currentThread();
      System.out.println("server entering critical section."+p.getName());
      for (i=0; i<MAXCOUNT; i++)
        {this.counter++;
        try {Thread.sleep(1);} catch(InterruptedException e) {e.printStackTrace();}
     for (i=0; i<MAXCOUNT; i++)
       {this.counter--;
       try {Thread.sleep(1);} catch(InterruptedException e) {e.printStackTrace();}}
     System.out.println("server leaving critical section."+p.getName()); }
```

RMI E THREAD SAFETY: RACE CONDITIONS

```
public int read() {
        return this.counter;}
public static void main(String [] args) {
     try {
         RMIThreadServerImpl localObject = new RMIThreadServerImpl();
         int port=Integer.parseInt(args[0]);
         LocateRegistry.createRegistry(port);
         Registry r=LocateRegistry.getRegistry(port);
          r.rebind("RMIThreadServer", localObject);
          }
catch(RemoteException e)
  {System.out.println("RemoteException"+e); }catch(Exception e)
           {System.out.println("Exception"+e);};
}}
```

RMI E THREAD SAFETY: RACE CONDITIONS

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class RMIThreadClient{


    public static void main (String[] args) {
       try {RMIThreadServer remObj;
            int port=Integer.parseInt(args[0]);
            Registry r=LocateRegistry.getRegistry(port);
            remObj = (RMIThreadServer) r.lookup("RMIThreadServer");
            System.out.println("client  before critical section."+remObj.read());
            remObj.update();
            System.out.println("client  after critical section."+remObj.read());
            } catch (Exception e) {System.out.println ("Client Exception"+e);
}}}
```

RMI E THREAD SAFETY: RACE CONDITIONS

```
> Java RMIThreadServerImpl
 server entering critical section.RMI TCP Connection(4)-192.168.202.149
 server entering critical section.RMI TCP Connection(6)-192.168.202.149
 server leaving critical section.RMI TCP Connection(4)-192.168.202.149
 server leaving critical section.RMI TCP Connection(6)-192.168.202.149
```

```
> Java RMIThreadClient
 client  before critical section.0
 client  after critical section.2250
```

```
> Java RMIThreadClient
 client  before critical section.1629
 client  after critical section.666
```