RMI vs LMI

- In questa lezione metteremo in risalto alcune differenze sostanziali tra programmazione locale (su una singola macchina) e *programmazione remota e distribuita* su più macchine.

- L'invocazione di metodo remoti (RMI) sebbene usi una sintassi simile a quella per l'invocazione di metodi locali (LMI) in realtà ha una semantica profondamente differente.

- La sintassi di una RMI ha grosso modo la seguente struttura:

```
try
{
 result = remoteObj.method(args);
}
catch (RemoteException e)
{
 // qui si gestisce l'eccezione remota
}
```

Rivediamo allora la semantica di LMI e poi definiamo quella di RMI per differenza.

Caratteristiche delle LMI

- Per poter eseguire un metodo di un oggetto di cui abbiamo una referenza dobbiamo fornire argomenti in numero e tipo appropriati.
- Un metodo può o meno avere la clausola "throws" per gestire eccezioni.
- Chi invoca un metodo deve gestire tramire l'istruzione "catch" le eccezioni dichiarate nella clausola "throws" del metodo. Ad esclusione di quelle eccezioni che il chiamante ha a sua volta dichiarato nella propria clausola "throws".
- Passaggio dei parametri e risultati di una invocazione LMI:
  - Tipi di dato primitivi vengono passati *per valore*;
  - Oggetti vengono passati *per referenza.*

- Le eccezioni vengono ritornate per referenza.
- Se il metodo termina normalmente allora è stato invocato (e quindi eseguito) esattamente una volta.
- Se un metodo ritorna un'eccezione non catturata da un'istruzione "catch" allora tale metodo è stato invocato una sola volta fino al punto in cui è stata emessa l'eccezione.
- Oggetti locali sono soggetti ad una garbage-collection standard.

Caratteristiche di RMI

RMI ha la stessa semantica di LMI eccetto per i seguenti punti:

- Un metodo remoto può essere invocato solo attraverso un interfaccia remota di cui tale metodo faccia parte.
- La dichiarazione di un metodo remoto all'interno di una interfaccia *deve* contenere la clausola "throws" per gestire eccezioni remote.

- I client di tali metodi remoti devono catturare e gestire le eccezioni remote attraverso il costrutto “catch”.

- Passaggio dei parametri e risultati di una invocazione RMI:

  - Tipi di dato primitivi vengono passati *per valore*;
  - Oggetti (locali) vengono passati per *deep copy*;
  - *Oggetti remoti esportati vengono passati per referenza remota*;
  - Oggetti remoti non esportati vengono passati per *deep copy*.

- Eccezioni emesse da una invocazione di un metodo remoto vengono passate per deep copy.

- `java.lang.RemoteObject` è una sottoclasse di `java.lang.Object` da cui eredita la semantica.

- RMI garantisce che quando un’invocazione remota termina (normalmente o attraverso un’eccezione) allora il metodo remoto è stato invocato “al più una volta”.

- Gli oggetti remoti sono soggetti ad una garbage-collection distribuita che precede quella locale.

Semantiche per LMI e RMI

- In LMI l'intero supporto di base (compilazione, istruzioni della JVM, esecuzione, etc) sono affidabili al 100% (a meno che non si arresti la macchina su cui è in esecuzione).

- Inoltre l'esecuzione di una LMI è sincrona. Vale a dire che lo stesso processore esegue sia il codice del metodo chiamato e sia il codice di chi ha invocato il metodo. Ciò vuol dire che per riprendere la propria esecuzione, il chiamante deve solo attendere che termini il metodo invocato.

- In RMI l'invocazione di un metodo è sicuramente meno affidabile. Ad esempio:

  1. Durante la comunicazione su rete certi pacchetti possono andare persi e devono essere ritrasmessi (fault-taulerance).

2. Certi pacchetti possono essere deliberatamente scartati dai routers per evitare congestionamenti.
3. In generale, RMI coinvolge due o più hosts che potrebbero fallire.
4. Infine, anche la porzione di rete che collega i due host può fallire; facendo erroneamente credere al client che la macchina remota sia fallita quando invece il problema è del collegamento di rete.

Fallimenti remoti

- Nei Client RMI non si possono ignorare i fallimenti remoti.
- Un client infatti deve gestire le eccezioni remote in maniera appropriata a seconda dell'applicazione che implementa.
- Si ricorda che un'eccezione remota può avere origine localmente o remotamente. Ad esempio:

- errori in fase di codifica/decodifica degli argomenti/risultati del metodo remoto
- errori di installazione
- errori transitori o permanenti della rete
- errori transitori o permanenti di Java
- errori transitori o permanenti del sistema delle risorse.

Fallimenti parziali

- Nel caso in cui emergano eccezioni remote non è facile capire se l'invocazione remota è fallita completamente o parzialmente.

- Ad esempio, un server remoto potrebbe aver aggiornato con successo un database a seguito di un'invocazione remota che però non termini correttamente per via di un fallimento nella comunicazione verso il client.

- Dal suo punto di vista il server immagina che l'invocazione remota abbia avuto successo mentre in realtà il client riscontra un fallimento in ricezione.

- Perciò i client RMI dovrebbero essere progettati in maniera tale da gestire anche fallimenti parziali.

- Questo è evidente quando si ha a che fare con operazioni su basi di dati remote.

- Riassumendo i fallimenti parziali (che differenziano LMI e RMI) dipendono dal fatto che le LMI usano il meccanismo di invocazione/risultato della JVM (che è molto affidabile) mentre RMI si affida alla rete (la quale è tutt'altro che affidabile).

Tempo di latenza

- I client di server remoti devono tener conto del tempo di latenza che intercorre da quando il metodo viene invocato a quando questo viene eseguito.

- Visto che il tempo di latenza è inevitabile, quanto tempo dovrebbe aspettare un client prima di sospettare un fallimento?
- La risposta a tale domanda dipende da un certo numero di fattori:
  - Banda di trasmissione della rete
  - Carico di lavoro della rete
  - Carico di lavoro del server
  - tempo stimato per l'esecuzione del server
  - ritardi nell'invocazione dovuti a RMI.
- Di defaut un client RMI aspetta indefinitamente che un metodo remoto ritorni un risultato.
- Comunque è possibile settare un timeout nel socket del client, in modo che se un metodo remoto prende più tempo del previsto viene messa una eccezione `InterruptedIOException` che si manifesta al client come una `UnmarshalException`.