Mobilità di codice

- Abbiamo già visto come un dato host possa trasmettere un oggetto (serializzabile) ad un altro host. Quest'ultimo che potrà eseguire l'oggetto a destinazione pur non possedendo il bytecode della classe relativa.

- Questo è possibile configurando nei server la proprietà

  `java.rmi.server.codebase`

  per definire il *codebase*.

- Il codebase consiste in una lista di uno o più URL, che denotano locazioni globali, da cui il client può caricare dinamicamente il codice (files *.class*) necessario per l'esecuzione, e non disponibile localmente attraverso il CLASSPATH.

- Come visto precedentemente, RMI usa Java Object Serialization per trasmettere oggetti non remoti. Ad esempio, ciò accade quando si invocano i metodi di un registro RMI passando gli stub.

- Nella serializzazione, Java RMI si prende cura di trasmettere, oltre agli oggetti, anche il il codebase delle classi relative.
- Più precisamente, quando viene trasmesso un oggetto, all'interno dello stream, viene settato il campo codebase del descrittore della classe con:
  - il valore della proprietà `java.rmi.server.codebase` della JVM che ha caricato per ultimo la classe dal suo CLASSPATH.
  - Oppure, il valore dell'URL da cui la classe è stata caricata dall'utente usando il metodo `RMICLassLoader.loadClass`, se è stato utilizzato tale metodo.
- Quando un oggetto viene *serializzato* in uno stream, il sistema RMI si prende cura di riportare il codebase nella descrizione della classe relativa invocando il metodo `ObjectOutputStream.annotateClass`.
- In fase di *de-serializzazione* dell'oggetto, se la sua classe non è disponibile localmente (attraverso il CLASSPATH del client), il ricevente tenterà di usare il codebase memorizzato nello stream per caricare dinamicamente la classe desiderata.

Formato di un codebase

- Un codebase è costituito da una lista di uno o più URL separati da spazi. Ciascun elemento della lista rispetta la seguente sintassi:

  `protocol://host[:port]/nomefile`

  dove:

  – `protocol` è il protocollo usato per lo scambio di informazioni. As esempio, http://, ftp://, file://, https://, ect.

  – `host` indica l'host presso cui si trova la risorsa.

  – `port` (opzionale) indica la porta TCP/IP tramite cui accedere al codebase (21 è la porta di default per ftp, 80 è la porta di default per http, ect....).

  – `file` denota la locazione all'interno dell'host della risorsa che si vuole caricare.

- Vediamo come viene ricercata una classe a partire da un codebase.

- Quando una classe, ad esempio `javarmi.hello.Hello`, non viene trovata dall'applicazione client nel suo CLASSPATH, si estrapola la struttura gerarchica della classe ottenendo per esempio `javarmi/hello/Hello`. A questo punto si scandiscono le URL del codebase e per ciascuna di queste si agisce nel seguente modo:
  - Se l'URL in questione ha la forma

    ```
    protocol://host[:port]/nomefile/
    ```

    con lo '/' finale, allora si assume che `nomefile` sia una directory e si cerca la classe desiderata al suo interno, eventualmente dentro un package, se si usano i packages.
  - Se invece l'URL in questione ha la forma

    ```
    protocol://host[:port]/nomefile
    ```

    senza lo '/' finale, si assume che `nomefile` sia un file JAR (ovvero un tar di files .class). In questo caso il file JAR è caricato nel cache ed ispezionato alla ricerca della classe desiderata.

- In entrambi i casi se la procedura di ricerca fallisce si passa all'esame del successivo URL all'interno del codebase.
- Se la classe desiderata non viene trovata in nessuno degli URL del codebase viene lanciata un'eccezione `ClassNotFoundException`.
- Se invece la classe viene trovata in uno degli URL indicati nel codebase, l'esecuzione continua.

VANTAGGI DELLA MOBILITÀ DI CODICE

- **(VERO) POLIMORFISMO**.
  - Nei linguaggi ad oggetti tale termine denota la possibilità di usare un oggetto di una sottoclasse B ogni qual volta che viene indicato di usare un oggetto di classe A, con $B < A$. [3]
  - Si noti che in generale questo non è possibile se il ricevente non conosce la sottoclasse B. In Java RMI, invece, ciò è possibile.

[3] Questo concetto è differente da quello che in Java è comunemente chiamato polimorfismo.

- Infatti, un'invocazione di un metodo remoto può ritornare un oggetto di una sottoclasse della classe indicata nell'interfaccia remota.
- Questo perchè in RMI se l'oggetto passato è stato serializzato appropriatamente ed annotato con un codebase contenente la sottoclasse, allora il destinatario, al momento della deserializzazione, caricherà la sottoclasse e l'applicazione verrà eseguita senza problemi.
- Un tale procedimento, è applicabile sia per quanto riguarda il passaggio dei risultati dal server al client, sia per quanto riguarda i parametri passati dal client al server.

- **SEMPLIFICAZIONE DEI CLIENTS.** Grazie alla mobilità di codice è possibile avere delle applicazioni client ridotte ai minimi termini che caricano dinamicamente il codice necessario a tempo di esecuzione.

- **MENO REINSTALLAZIONI.** Usando il caricamento dinamico delle classi non è necessario la reinstallazione di clients le cui classi cambiano durante l'applicazione. È sufficiente aggiornare una singola copia della classe presso il codebase del server. È il sistema RMI che si prende cura di propagare tale modifica.

- **MENO INCONSISTENZE** Se i client condividono la medesima classe presso il codebase del server non vi saranno inconsistenze, ovvero client che usano versioni differenti della medesima classe.

Considerazioni sulla sicurezza

- La mobilità di codice pone ovvi problemi di sicurezza sul contenuto del codice che viene caricato.
- Per tale ragione, il caricamento di codice attraverso *codebase* è abilitato solo in presenza di un *Java Security Manager*.
- Il Java Security Manager è un meccanismo che si occupa di gestire la sicurezza di un'applicazione con particolare riguardo per il caricamento dinamico di codice.

- La configurazione di default del Java Security Manager garantisce completa sicurezza. Più precisamente, ogni permesso richiesto dall'applicazione deve essere indicato esplicitamente.

- Ad esempio, di default, le classi caricate dinamicamente non possono interagire col file-system locale, la rete, o l'interfaccia grafica.

- Comunque è possibile configurare il security manager in modo da fornire permessi differenti a classi caricate da codebase differenti. Dando più diritti a quelle classi provenienti da "codebase fidati" e meno a quelli provenienti da altri codebase.

Adesso vediamo i passi per effettuare correttamente il caricamento dinamico di classi tramite codebase.

IMPORTANTE: Si noti che il caricamento dinamico di una classe attraverso codebase può essere fatto sia dal client, se riceve come risultato un oggetto serializzato di cui non conosce la classe, e sia dal server, se riceve tra i parametri di un'invocazione remota un oggetto serializzato di cui non conosce la classe.

Proprietà del codebase

- Un server (o un client), che desidera mettere a disposizione delle classi, deve essere lanciato settando appropriatamente la proprietà di sistema `java.rmi.server.codebase`.

- In presenza di tale proprietà, quando si serializza un oggetto, la descrizione della classe relativa viene annotata con il codebase. Si noti che se un server viene lanciato con la proprietà di codebase allora esso continua a caricare le classi dal CLASSPATH ma tali classi quando trasmesse per la serializzazione verranno annotate col codebase in modo da poter essere caricate da client remoti.

- In alternativa all'utilizzo della proprietà di codebase il client (o server) può caricare dinamicamente (da programma) una classe da una URL attraverso il metodo `RMIClassLoader.loadClass`. In tal caso la classe trasmessa verrà annotata con l'URL utilizzata.

- Questa tecnica è più flessibile poichè consente di usare codebase differenti per caricare classi differenti. È necessario che le classi caricate in questo modo non siano accessibili attraverso il CLASSPATH, altrimenti la classe non verrà annotata col codebase usato dalla `loadClass` ma con l'eventuale codebase con cui è stata lanciata l'applicazione.

Registri RMI e codebase

La mobilità di codice può essere fonte di subdoli errori collegati all'inizializzazione dei registri RMI. Vediamone alcuni:

1. Al momento del lancio del server viene emessa una

   `ClassNotFoundException`

   riguardante la classe stub. Tale eccezione viene lanciata al momento dell'esecuzione della bind (o rebind) nel caso in cui:

   (a) Il registro RMI sia stato preventivamente lanciato in un environment in cui ne `java.rmi.server.codebase` ne il CLASSPATH consenta al registro di accedere alle classi stub.

(b) La proprietà `java.rmi.server.codebase` con cui è stato lanciato il server o è assente oppure non consente di accedere agli stub.

In tal caso, infatti, il metodo bind passerà al registro RMI lo stub senza l'annotazione del codebase da cui caricarlo (per il punto 1b). Di conseguenza, nelle ipotesi indicate al punto 1a, il registro RMI incontrerà una `ClassNotFound Exception` nel momento in cui deserializza l'oggetto stub poichè non riesce a ricavare la classe relativa né dal CLASSPATH e né dal codebase che avrebbe dovuto inviargli il server insieme allo stub.

2. Al momento del lancio del server viene emesso un errore

`NoClassDefFoundError`

avente origine nella bind e riguardante l'interfaccia remota. Per evitare tale errore il registro deve essere lanciato in modo da poter aver accesso alla classe dell'interfaccia remota. Oppure, in alternativa, l'interfaccia remota deve essere contenuta nel codebase utilizzato per lanciare il server.

3. Al momento del lancio del client viene emessa, durante l'esecuzione della lookup, una `ClassNotFoundException` relativa alla classe stub. Le cause di ciò possono essere due:

    - Indipendentemente dal codebase del server, il registro RMI potrebbe essere stato lanciato con accesso alle classi stub. In tal caso durante l'invocazione della bind il registro RMI riceve dal server lo stub correttamente annotato col codebase. Ma tale codebase, sebbene presente nello stream, non viene utilizzato poichè il registro trova la classe stub nel suo CLASSPATH. Di conseguenza, al momento di spedire una copia dello stub al client che ha invocato la lookup, il registro annoterà lo stub con il proprio codebase (quello con cui è stato lanciato il registro, non il server) che verosimilmente non è stato settato. E quindi il client riceverà uno stub senza codebase ed incontrerà una `ClassNotFoundExeception`.
    - Lo stesso problema menzionato sopra sorge quando il server sia stato lanciato con codebase scorretto ed il registro RMI carica la classe stub dal suo CLASSPATH.

Protocolli per il codebase

Come accennato prima possiamo usare differenti protocolli per il codebase:

1. **http://** Ciò richiede la presenza di un server http. Possiamo poi settare il codebase con un'area accessabile dall'esterno, come ad esempio la directory Linux *public_html* utilizzando la propria homepage come spazio accessibile dall'esterno.

2. **file://** Tale protocollo funziona solo se tutti i partner coinvolti condividono il medesimo file system (come accade a voi in laboratorio). Non funziona se l'URL è espressa in termini del file system locale dell'utente (Es. `file://~/javarmi/hello`). Gli indirizzi infatti devono essere assoluti perchè utilizzati da client differenti.

3. **ftp://** simile ad http, ma richiede un server ftp in esecuzione presso la locazione del codebase. Si noti che ftp non è sempre supportato in presenza di firewall.

4. **https://** simile ad http, ma con l'utilizzo di socket sicuri SSL.

Caricamento dinamico del codice client

- Java RMI dà la possibilità di caricare quasi per intero l'applicazione client dal codebase del server.

- Ovvero il codice del client si limita ad un semplice programma di bootstrap che carica dinamicamente la classe dell'applicazione che deve rispettare una certa interfaccia.

- Per semplicità un'applicazione client può essere un'implementazione dell'interfaccia `java.lang.Runnable`. Ad esempio:

```
public class Client implements Runnable { .......}
```

- In tal modo il programma di bootstrap presso il client dovrà solamente caricare il codice della sua applicazione e mandarlo in esecuzione col metodo `run`.

- Ovviamente è necessario che tale classe non sia disponibile nel CLASSPATH del client, altrimenti non verrebbe caricata dal codebase.

In questo corso vedremo due possibili schemi di bootstrap per il client:

1. `RMIClassloader` **bootstrap**

   Con questa tecnica viene utilizzato il metodo `RMIClassloader.loadClass` per caricare dinamicamente, da un certo URL, il codice del client.

```
public class URLClientBootstrap {
 static String codebase=.........; // codebase URL
 static String clientClass=......;
 //nome della classe che si vuole caricare
 public static void main(String[] args) throws Exception {
  System.setSecurityManager(new SecurityManager);
  Class classClient = RMIClassLoader.loadClass(codebase,
                                               clientClass);
  Runnable client = (Runnable)classClient.newInstance();
  client.run();
 }
}
```

In una applicazione più realistica le variabili 'codebase' e 'clientClass' andrebbero acquisite da linea di comando con relativa gestione delle eccezioni.

2. **Bootstrap via RMI**

   Con questa tecnica si acquisisce il codice da un server remoto di bootstrap registrato in un registro RMI. Uno dei metodi di tale server viene invocato dal client per ottenere un oggetto runnable della classe richiesta. Per poter serializzare correttamente l'oggetto di classe runnable il registro RMI deve essere lanciato con una proprietà di `java.rmi.server.codebase`. Ad esempio:

```java
//Interf. remota per il servizio di bootstrap.
public interface Bootstrap extends java.rmi.Remote {
  public Runnable getClient() throws RemoteException;
}

// Bootstrap del client.
public class RMIClientBootstrap {
 static String bootstrapServer=.....;
 public static void main(String[] args) throws Exception {
  System.setSecurityManager(new RMISecurityManager());
  Bootstrap bs = (Bootstrap)Naming.lookup(bootstrapServer);
  Runnable client = bs.getClient();
  client.run();
 }
}
```