Java Remote Method Invocation (RMI)

Java RMI è un'implementazione Java-specific del modello RPC, ovviamente esteso per considerare un'architettura a oggetti.

In modo analogo a RPC, l'obbiettivo di Java RMI è quello di rendere il più possibile simili la chiamata locale e la chiamata remota.

La piattaforma Java facilita la gestione dell'eterogeneità dei nodi e consente di mascherare completamente le differenze sintattiche fra chiamata locale e remota:

- uso di interfacce Java al posto di IDL;
- uso di tipi di dati primitivi Java;
- serializzazione per trasmissione (dello stato) degli oggetti;
- repository di implementazioni per condivisione del codice.

Modello a oggetti distribuiti

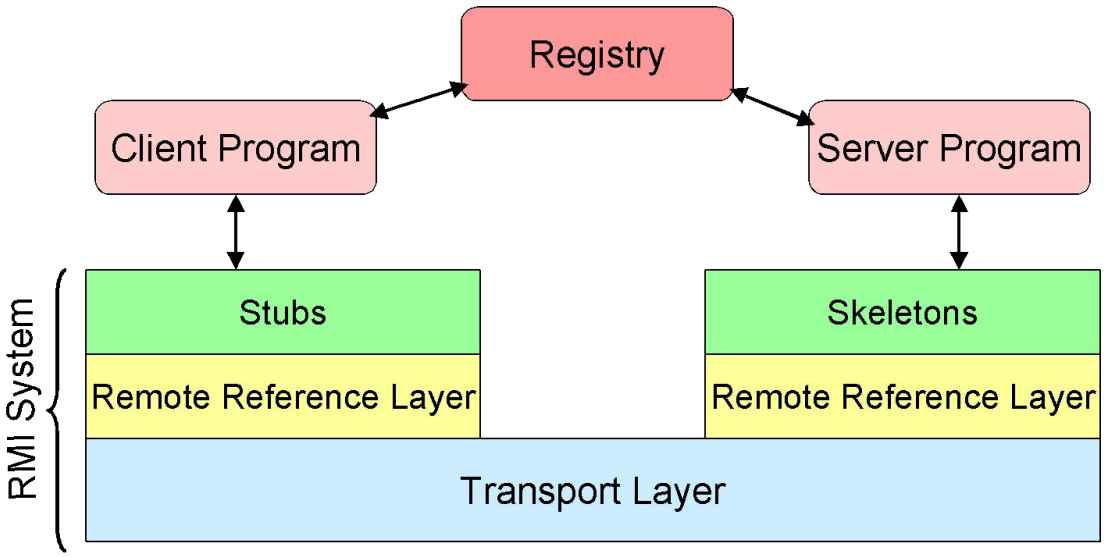
I sistemi a oggetti distribuiti, come Java RMI e CORBA, permettono di realizzare applicazioni basate sul paradigma a oggetti

Essi presentano numerosi vantaggi, in termini di estendibilità, facilità di integrazione e semplicità di sviluppo, e si sono dimostrati vincenti a livello enterprise

Nel modello a oggetti distribuito di Java RMI un *oggetto remoto* consiste in un oggetto:

- con metodi invocabili da un'altra JVM, in esecuzione su un host differente
- descritto **tramite interfacce remote** che dichiarano i metodi accessibili da remoto

Architettura RMI



Architettura RMI

Stub: proxy locale su cui vengono fatte le invocazioni destinate all'oggetto remoto

Skeleton: elemento remoto che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sul server

Remote Reference Layer: fornisce il supporto alle chiamate inoltrate dallo stub. Il RRL ha il compito di instaurare la connessione fra il client e il server eseguendo operazioni di codifica e decodifica dei dati

Transport Layer: localizza il server RMI relativo all'oggetto remoto richiesto, gestisce le connessioni (TCP/IP, timeout) e le trasmissioni (sequenziali, serializzate)

Registry: servizio di naming che consente al server di pubblicare un servizio e al client di recuperarne il proxy

Sviluppo di un'applicazione RMI

Per realizzare un'applicazione distribuita usando Java RMI dobbiamo:

1. Definire l'interfaccia di servizio
2. Progettare le implementazioni dei componenti utilizzabili in remoto e compilare le classi necessarie (con **javac**)
3. Generare stub e skeleton delle classi utilizzabili in remoto (con il compilatore RMI **rmic**)
4. Attivare il registry dei servizi
5. Registrare il servizio (il **server** deve fare una **bind sul registry**)
6. Effettuare il **lookup sul registry** (il **client** deve ottenere il reference all'oggetto remoto)

A questo punto l'interazione tra il client e il server può procedere

Definizione dell'interfaccia di servizio

L'interfaccia del servizio è il contratto che deve essere rispettato dai servitori e anche dai clienti (a cui deve essere nota).

Le interfacce di servizio sono semplici interface Java che devono ereditare dall'interfaccia **java.rmi.Remote**.

Tutti i metodi pubblici di queste interfacce saranno invocabili da remoto. Tali metodi devono dichiarare di sollevare **java.rmi.RemoteException**.

```
import java.rmi*;
public interface EchoService extends Remote {
  public String getEcho(String echo)
         throws RemoteException;
}
```

Implementazione Server

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

public class EchoServiceImpl extends UnicastRemoteObject
                              implements EchoService {

    public String getEcho(String str) throws
                                        RemoteException {
        return str;
    }

    public EchoServiceImpl() throws RemoteException {
        super();
    }
```


Implementazione Server

```
    public static void main(String[] args) {
        try {
            EchoService service = new EchoServiceImpl();
            Naming.rebind("EchoService", service);
        } catch(Exception e) {
            System.err.println(e.getStackTrace());
            System.exit(1);
        }
    }
}
```

Il server deve estendere la classe UnicastRemoteObject e implementare costruttore e metodi dell'interfaccia remota

Implementazione Client

```
import java.io.*;
import java.rmi.*;

public class EchoServiceClient
{
    public static void main(String[] args) {
        try {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader stdIn = new BufferedReader(isr);

            /* Connessione al registry RMI remoto: si deve
               ricercare l'interfaccia e ritirarla in modo
               corretto */
            EchoService service =
                (EchoService)Naming.lookup("EchoService");

            // Interazione con l'utente
            System.out.print("Messaggio? ");
            String message = stdIn.readLine();
```


Implementazione Client

```
            // Richiesta del servizio remoto
            String echo = service.getEcho(message);

            // Stampa risultati
            System.out.println("Echo: " + echo + "\n");
        }
        catch (Exception e) { e.printStackTrace(); System.exit(1); }
    }
}
```

Compilazione e creazione stub e skeleton

Compilazione di interface, client e server:

```
[mauro@remus rmi]$ javac EchoService.java
EchoServiceClient.java EchoServiceImpl.java
```

Creazione di Stub e Skeleton con il compilatore RMI:

```
[mauro@remus rmi]$ rmic –v1.1 EchoServiceImpl
```

rmic genera EchoServiceImpl_Stub.class e EchoServiceImpl_Skel.class

Compilazione e creazione stub e skeleton

1) Avviamento del registry:

```
[mauro@remus rmi]$ rmiregistry
```

2) Avviamento del server:

```
[mauro@remus rmi]$ java EchoServiceImpl
```

3) Avviamento del client:

```
[mauro@remus rmi]$ java EchoServiceClient
```

Serializzazione

In Java marshalling e unmarshalling vengono realizzati tramite il meccanismo di *serializzazione* offerto dal linguaggio.

La serializzazione permette la lettura/scrittura di dati complessi da/su stream (es. file o socket) e viene fatta in maniera **trasparente** dal supporto (ovverosia dal tipo di stream).

Serializzazione: trasformazione di oggetti complessi in semplici sequenze di byte su uno stream di output
=> metodo **writeObject()**

Deserializzazione: decodifica di una sequenza di byte e costruzione di una copia dell'oggetto originale da uno stream di input
=> metodo **readObject()**

Serializzazione

In Java la serializzazione è utilizzata in molti casi:

- persistenza
- trasmissione di oggetti tra macchine diverse (parametri e valori di ritorno in RMI)

Sono **automaticamente serializzabili** (tramite l'apposito supporto fornito dalla Java Virtual Machine) istanze di oggetti che:

- **implementano l'interfaccia *Serializable***
- contengono **esclusivamente oggetti** (o riferimenti a oggetti) **serializzabili**

Infatti, la serializzazione in Java è ricorsiva.

ATTENZIONE: non viene trasferito l'oggetto vero e proprio ma solo le informazioni che ne caratterizzano l'istanza => no metodi, no costanti, no variabili statiche, no variabili transient.

Serializzazione

La maggior parte delle classi della libreria Java sono già automaticamente serializzabili, ma alcune non lo sono (es. File).

Se una classe contiene oggetti non serializzabili, deve dichiararli come ***transient***, e implementare i metodi ***readObject*** e ***writeObject***.

Al momento della deserializzazione sarà **ricreata una copia** dell'istanza "trasmessa", usando il codice (ovverosia, il file .class - che deve quindi essere accessibile!!!) dell'oggetto e le informazioni ricevute.

Esempio di persistenza tramite serializzazione

Serializzazione oggetto record su file *data.ser*:

```
Record record = new Record();
FileOutputStream fos = new
FileOutputStream("data.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(record);
```

Deserializzazione oggetto da file *data.ser*:

```
FileInputStream fis = new
FileInputStream("data.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
record = (Record)ois.readObject();
```

Passaggio di parametri

Il passaggio dei parametri **dipende dal tipo di parametro in considerazione**

Tipo	Metodo Locale	Metodo Remoto
Tipi primitivi	Per valore	Per valore
Oggetti	Per riferimento	Per valore (serializzazione)
Oggetti Remoti Esportati	Per riferimento (all'oggetto Stub)	Per riferimento remoto

Passaggio per valore => Serializable Objects

Passaggio per riferimento => Remote Objects

Passaggio di parametri

Oggetti serializzabili

Oggetti la cui locazione non è rilevante per lo stato.

Sono passati **per valore**: ne viene serializzata l'istanza che sarà deserializzata a destinazione per crearne una copia locale.

Oggetti remoti

Oggetti la cui funzione è strettamente legata alla località in cui eseguono (server).

Sono passati **per riferimento**: ne viene serializzato lo stub, creato automaticamente dal proxy (stub o skeleton) su cui viene fatta la chiamata in cui compaiono come parametri

Localizzazione del servizio

Un client in esecuzione su una macchina ha bisogno di localizzare un server a cui vuole connettersi, che è in esecuzione su un'altra macchina.

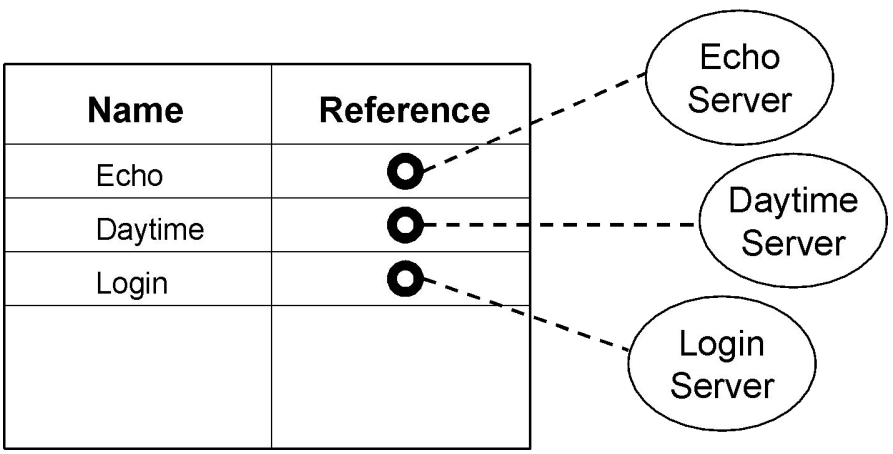
Tre possibili soluzioni:

- Il client conosce in anticipo dov'è il server
- L'utente dice all'applicazione client dov'è il server (es. e-mail client)
- Un servizio standard (***naming service***) in una locazione ben nota, che il client conosce, funziona come *punto di indirezione*

Java RMI utilizza un naming service: ***RMI Registry***

Localizzazione del servizio

Il registry RMI mantiene un insieme di coppie **{name, reference}**, dove *name* è una stringa arbitraria non interpretata e *reference* è un riferimento a un oggetto remoto



Localizzazione del servizio

Per accedere ai servizi del registry RMI si usa la classe ***java.rmi.Naming***, che fornisce i seguenti metodi (statici):

```
public static void bind(String name, Remote obj)
public static void rebind(String name, Remote obj)
public static void unbind(String name)
public static String[] list(String name)
public static Remote lookup(String name)
```

Il parametro name è una URI che combina la locazione del registry e il nome logico del servizio, nel formato:

[rmi://hostname:port/]object_name

La porta di default è la 1099.

Sicurezza del registry

Problema: accedendo al registry (individuabile interrogando tutte le porte di un host) è possibile ridirigere per scopi maliziosi le chiamate ai server RMI registrati (es. **list()+rebind()**)

Soluzione: i metodi bind(), rebind() e unbind() sono invocabili **solo dall'host su cui è in esecuzione il registry**

-> non si accettano modifiche della struttura client/server dall'esterno

Sull'host in cui vengono effettuate le chiamate al registry deve essercene almeno uno in esecuzione

Sicurezza del registry

Ogni JVM definisce ambienti di esecuzione differenziati e protetti per diverse parti, in particolare per quanto riguarda le interazioni con applicazioni distribuite e l'uso di codice remoto.

Per rendere maggiormente sicure le applicazioni RMI è necessario impedire che le classi scaricate da host remoti effettuino operazioni per le quali non sono state preventivamente abilitate. A questo fine, si usano **ClassLoader** che effettuano il controllo sul caricamento di codice remoto, associati a **SecurityManager** per la specifica di protezione.

Java mette a disposizione un SecurityManager per applicazioni RMI: **RMISecurityManager**.

Serialization Filtering

La deserializzazione di dati non attendibili, in particolare da un Client sconosciuto, non attendibile o non autenticato, è un'attività intrinsecamente pericolosa perché il contenuto del flusso di dati in ingresso determina gli oggetti creati, i valori dei relativi campi e i riferimenti tra di essi. Mediante un'attenta costruzione del flusso, un avversario può eseguire codice in classi arbitrarie con intenti dannosi.

La serializzazione in Java è stata la causa di **molti gravi problemi di sicurezza**:
https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=java+serialization

Necessità di **implementare comunicazioni su canale cifrato e di adottare serialization filtering**:
https://docs.oracle.com/en/java/javase/19/core/serialization-filtering1.html
https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

Security Manager - Deprecato!

In passato il meccanismo principale per la gestione della sicurezza delle applicazioni RMI era il Security Manager. Tuttavia esso adotta un approccio che si è rivelato inadeguato, ed è stato recentemente deprecato: https://openjdk.org/jeps/411

RMISecurityManager era pensato per essere istanziato al lancio dell'applicazione RMI, sia nel client che nel server:

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
}
```

Se il SecurityManager non è specificato, non è permesso nessun caricamento di codice remoto da parte sia del client (Stub) che del server

Security Manager - Deprecato!

I Security Manager erano pensati per effettuare l'enforcing delle politiche di sicurezza nelle applicazioni Java.

Le politiche di sicurezza sono definite dagli sviluppatori, di solito usando dei file di policy.

Sia il server che il client devono essere lanciati specificando il file con le autorizzazioni che il security manager deve caricare.

Esempio:

```
java -Djava.security.policy=echo.policy EchoServiceImpl
java -Djava.security.policy=echo.policy EchoServiceClient host
```

SM: Esempio file di policy - Deprecato!

```
grant {
  permission java.net.SocketPermission
    "*:1024-65535",
    "connect, accept, resolve";
  permission java.net.SocketPermission
    "*:80", "connect";
  permission java.io.File Permission
    "c:\\home\\RMIdir\\-", "read";
};
```

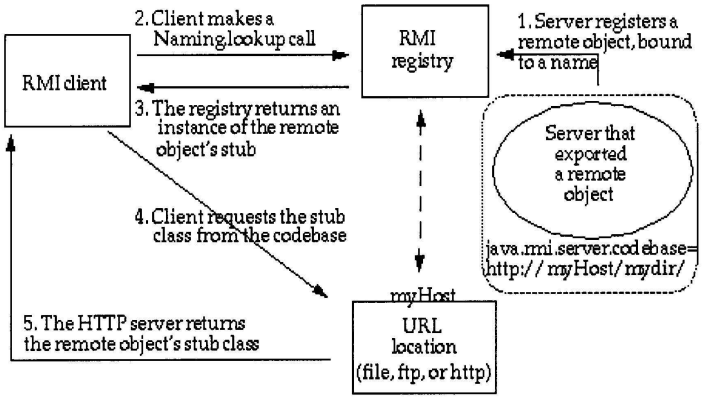
Il primo permesso consente al Client e al Server di instaurare le connessioni necessarie all'interazione remota.

Il secondo e il terzo permesso consentono rispettivamente di prelevare il codice da un server HTTP e di accedere in lettura ad una directory del file system locale.

Condivisione codice - Insicuro!

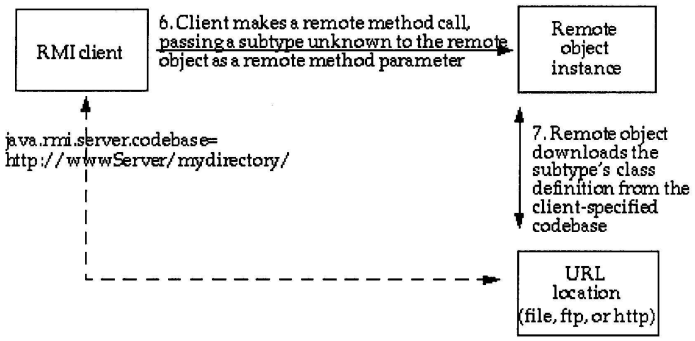
RMI originariamente permetteva a Client e Server di condividere codice attraverso il meccanismo del codebase.

Il codebase viene usato dal **Client** per scaricare le classi necessarie del server (interfaccia, stub, oggetti restituiti come valori di ritorno)



Condivisione codice - Insicuro!

Il codebase viene usato dal **Server** per scaricare le classi necessarie relative al client (oggetti passati come parametri nelle chiamate)



Attenzione: la condivisione del codice rappresenta un meccanismo estremamente insicuro, che viene qui discusso solo per ragioni di interesse storico.

Garbage collection di oggetti distribuiti

In un sistema a oggetti distribuito, è desiderabile la deallocazione automatica degli oggetti remoti che non hanno più nessun riferimento presso dei client.

Il sistema RMI utilizza un algoritmo di garbage collection distribuito basato sul conteggio dei riferimenti:

- Ogni JVM aggiorna una serie di contatori ciascuno associato ad un determinato oggetto.
- Ogni contatore rappresenta il numero dei riferimenti ad un certo oggetto che in quel momento sono attivi su una JVM.
- Ogni volta che viene creato un riferimento ad un oggetto remoto il relativo contatore viene incrementato. Per la prima occorrenza viene inviato un messaggio che avverte l'host del nuovo client.
- Quando un riferimento viene eliminato il relativo contatore viene decrementato. Se si tratta dell'ultima occorrenza un messaggio avverte il server.

Esempio di codice in Java RMI - Interfaccia

In questo esempio viene riportato l'implementazione del servizio di **RemoteLs** in Java RMI.

Definizione **dell'interfaccia di servizio:**

```
import java.rmi.*;
import java.util.List;
public interface LsService extends Remote {
    public List<String> ls(String directory)
                  throws RemoteException;
}
```


Esempio di codice in Java RMI – Server (1/2)

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

public class LsServiceImpl extends UnicastRemoteObject
implements LsService {

public LsServiceImpl() throws RemoteException { super(); }

public List<String> ls(String directory) throws RemoteException
{
    File dir = new File(directory);
    String files[] = dir.list();
    return Arrays.asList(files);
 }
```

Esempio di codice in Java RMI – Server (2/2)

```
public static void main(String[] args) {
      try {
          LsService service = new LsServiceImpl();
       Naming.rebind("LsService", service);

      } catch(RemoteException e) {
    System.err.println(e.getStackTrace());
       System.exit(1);
}
```


Esempio di codice in Java RMI – Client (1/2)

```
import java.rmi.*;
…
public class LsServiceClient
{
    public static void main(String[] args) {
    try {
           BufferedReader stdIn = new
       BufferedReader(new InputStreamReader(System.in));

/* Connessione al registry RMI remoto */

LsService service = (LsService)Naming.lookup("LsService");

// utente
System.out.println("Inserire il nome della directory:");
String dirname = stdIn.readLine();
```

Esempio di codice in Java RMI – Client (2/2)

```
    //Chiamata del servizio

    List<String> list = service.ls(dirname);

    System.out.println("Lista file directory "+dirname+":");

    for(String file: list) {
      System.out.println(file);
    }

    }
    catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
}
```