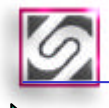
CEFRIEL
Consorzio per la Formazione e la Ricerca
in Ingegneria dell'Informazione

Politecnico
di Milano

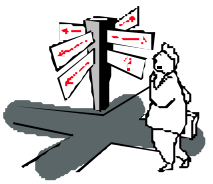
Programmazione Object Oriented in Java

Java Remote Method Invocation

Docente:
Diego Peroni
CEFRIEL
peroni@cefriel.it



Indice degli argomenti



Introduzione

- RMI si pone come obiettivo quello di facilitare la comunicazione fra **applicazioni Java** residenti su macchine remote, presupponendo un ambiente **omogeneo** Java
- RMI fornisce al programmatore una tecnologia molto simile al concetto di **Remote Procedure Call**.
  - Realizzazione di applicazioni distribuite ad oggetti in linguaggio Java
  - Permette di sfruttare al pieno le caratteristiche di Java anche nella programmazione distribuita
- A differenza di RPC utilizza il paradigma **Object Oriented**
  - Il client invoca metodi di oggetti che appaiono locali (sulla stessa *Java Virtual Machine*) ma in realtà sono repliche di oggetti presenti fisicamente su JVM remote
  - RMI, proposto dalla Sun, costituisce sostanzialmente l'alternativa a CORBA.

Gli obiettivi di RMI

- Supportare l'invocazione di oggetti remoti fra VM diverse
- Supportare le *callback* tra server e client
- Integrare il modello distribuito nel linguaggio Java conservandone la semantica
- Definire in modo chiaro le differenze fra oggetti locali e remoti
- Rendere la progettazione di applicazioni distribuite quanto più semplice possibile
- Rispettare la sicurezza dell'ambiente *runtime* di Java
- Permettere molti meccanismi di invocazione di oggetti remoti (ad es. invocazione singola di un oggetto o invocazioni multiple di un oggetto replicato su più locazioni)

Gli obiettivi di RMI (2)

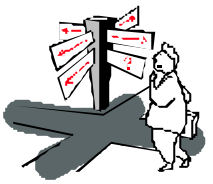
- Fornire un sistema di gestione della memoria distribuito (*Distributed Garbage Collector*) per disallocare automaticamente oggetti remoti
- Possibilità di supportare diversi meccanismi (protocolli) di trasporto per le invocazioni remote
- Estendere le caratteristiche di Java (sicurezza e caricamento dinamico delle classi) nella comunicazione fra Java VM distribuite
- Trasmettere le eccezioni sollevate durante le invocazioni di metodi remoti alle varie JavaVM coinvolte nella comunicazione

Definizioni:

- **Oggetto remoto**: oggetto i cui metodi possono essere invocati da una *Java Virtual Machine* diversa da quella in cui l'oggetto risiede
- **Interfaccia remota**: interfaccia Java che dichiara quali sono i metodi che possono essere invocati da una diversa *Java Virtual Machine*
- **Server**: uno o più oggetti remoti che, implementando una o più interfacce remote, offrono delle risorse (dati e/o procedure) a macchine esterne distribuite sulla rete
- ***Remote Method Invocation* (RMI)**: invocazione di un metodo presente in una interfaccia remota implementata da un oggetto remoto. La sintassi di una invocazione remota è identica a quella locale

Indice degli argomenti

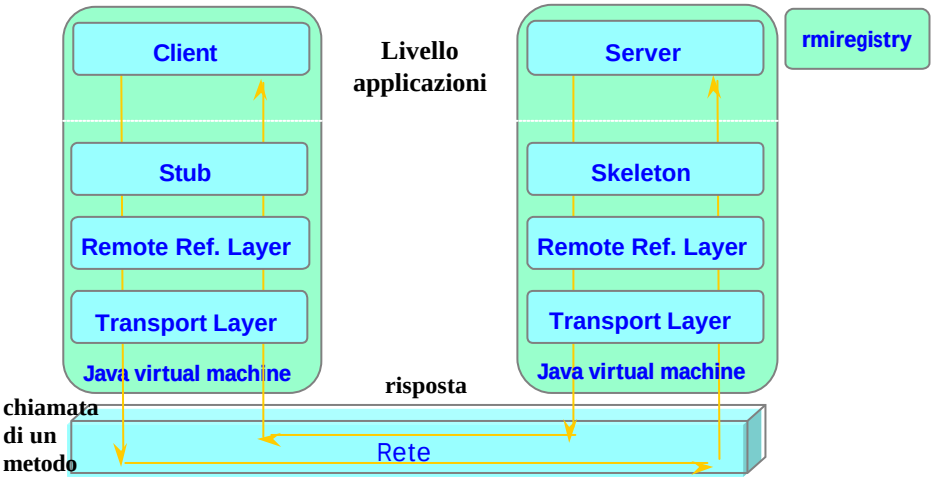
- Contesto e motivazioni
- Architettura e concetti di base
  - Componenti di RMI
  - RMIRegistry
  - Interfacce, eccezioni e classi
- Lo sviluppo di una applicazione
- L'esecuzione di una applicazione
  - Caricamento dinamico delle classi
  - RMI e sicurezza
- Il passaggio dei parametri
- L'utilizzo del registry



Architettura di RMI

- Client
  - Richiede i servizi offerti dai server
- Server
  - E' il fornitore dei servizi richiesti dai client
- RMIRegistry
  - Servizio di Naming: consente di associare un identificatore simbolico al riferimento effettivo ad un oggetto remoto
  - Viene utilizzato dal client in fase di bootstrap per recuperare il riferimento al "primo" oggetto remoto
- NOTA: i ruoli di client e di server non sono rigidamente fissati e valgono nel contesto di una singola interazione

Architettura di RMI



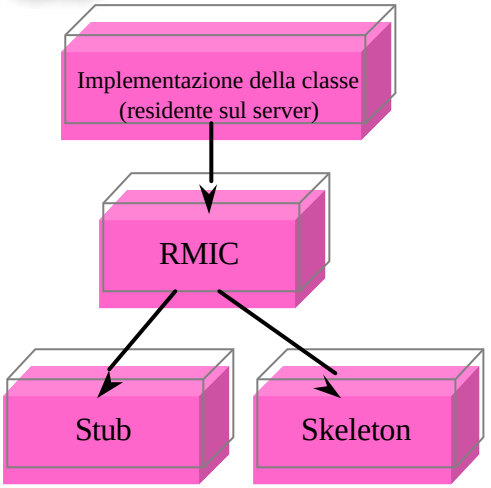
Le componenti di RMI

- Una invocazione RMI parte dal livello applicativo client, attraversa i vari strati di RMI fino ad arrivare al trasporto. Non appena la chiamata arriva sulla JavaVM del server, viene fatta risalire attraverso i vari strati di RMI per arrivare all'opportuno oggetto Java che implementa il servizio richiesto.
- La risposta segue la strada inversa: parte dallo *skeleton*, passa nel *remote reference layer* del server, nel suo strato di trasporto ed arriva alla macchina client, dove risale fino all'applicazione.

Le componenti di RMI

- Lo strato *stub/skeleton*
  - Lo *stub* è il riferimento locale che il client possiede dell'oggetto remoto. Lo *stub* inoltra le richieste del client all'opportuna JavaVM
  - Lo *skeleton* riceve le invocazioni dallo strato inferiore e le smista all'opportuno oggetto Java che implementa il metodo richiesto
- Lo strato di riferimento remoto si occupa di gestire la semantica della chiamata
  - Un primo compito riguarda la comunicazione verso lo strato inferiore dell'apertura di connessioni verso un singolo host se l'oggetto remoto è *unicast* o verso più host se l'oggetto remoto è m*ulticast*
  - Si occupa anche di nascondere agli strati superiori le differenze fra il caso in cui il server è costantemente in esecuzione su una macchina dal caso in cui il server si attiva nel momento in cui viene fatta una chiamata
- Lo strato di trasporto provvede all'instaurazione ed alla gestione delle connessioni verso gli host remoti

Strato *Stub/Skeleton*



- A partire dalla classe che implementa il servizio, tramite il "compilatore" *rmic* è possibile generare lo *stub* e lo *skeleton* relativi al servizio stesso
- Lo *stub* ora risiede sul server: verrà caricato automaticamente sul client dall'*AppletClassLoader* o dall'*RMIClassLoader*
- Il sistema Java carica da remoto solo le classi non disponibili localmente

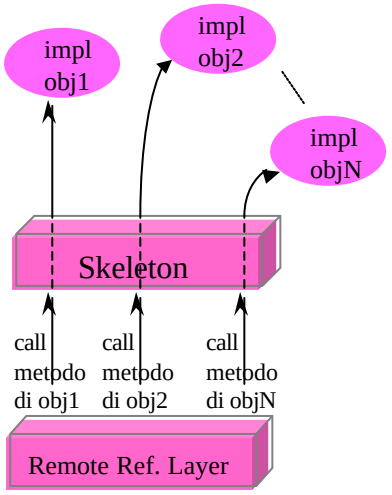
Architettura di RMI: Stubs e Marshalling dei parametri

- Per il client l'invocazione di un metodo su un oggetto remoto non è diversa da una chiamata di metodo su un oggetto locale
- La chiamata viene ricevuta da un oggetto surrogato dell'oggetto server, chiamato "stub" e attivo sulla macchina client, che si occupa di
  - Iniziare una connessione con la Java Virtual Machine remota
  - Convertire i parametri della chiamata di metodo in un formato adatto alla trasmissione in rete (Parameter Marshaling) e trasmettere i parametri serializzati
  - Attendere il risultato della chiamata
  - Convertire il risultato dell'operazione (o l'eventuale eccezione ricevuta) e restituire il risultato al client
- Lo stub maschera la serializzazione dei parametri ed i dettagli di basso livello della comunicazione, in modo da presentare al client un meccanismo di invocazione semplificato

Architettura di RMI: Skeletons

- Nella Java Virtual Machine Remota, ogni oggetto remoto può avere un corrispondente oggetto skeleton
  - Uno skeleton è responsabile del dispatching della chiamata all'oggetto che implementa l'interfaccia remota
  - Negli ambienti interamente basati su piattaforma Java 2 v1.2 gli skeleton non sono più necessari
    - Viene utilizzato del codice "generico", ovvero non specializzato per un particolare oggetto server

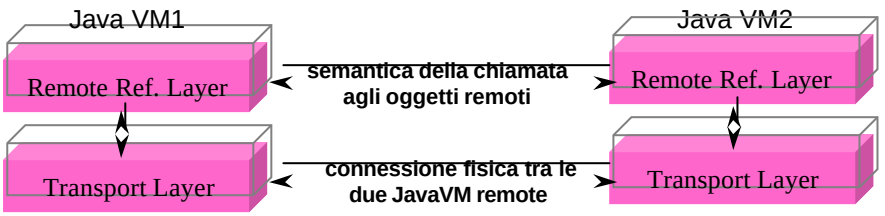
Architettura di RMI: Skeletons - Funzionalità



- E' responsabile di indirizzare ogni chiamata remota verso l'oggetto che implementa il metodo corrispondente
- Effettua l'*unmarshalling* dei parametri, passando da *stream* agli oggetti Java corrispondenti
- Effettua il *marshalling* dei valori di ritorno (o delle eccezioni) dei metodi remoti invocati.

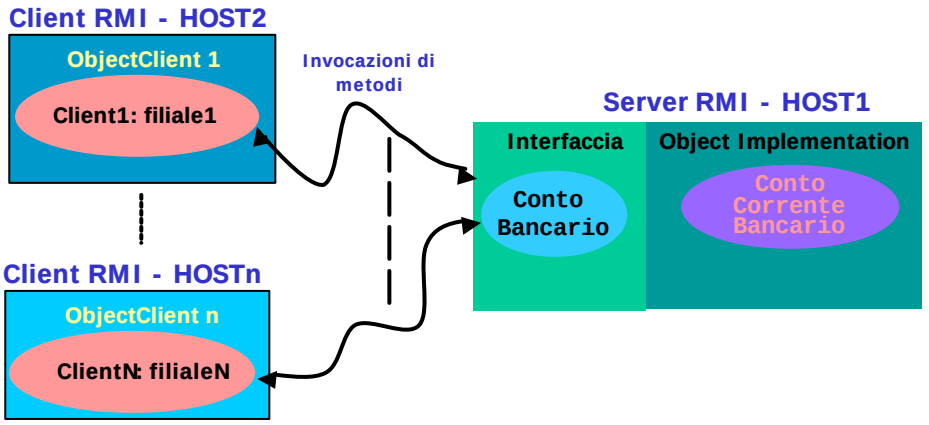
NB: il server, nell'implementare un servizio, potrebbe avvalersi di altri oggetti che potrebbero essere a loro volta remoti.

Architettura di RMI: Remote Reference Layer



- Questo strato definisce la semantica delle chiamate agli oggetti remoti
- Ogni implementazione di oggetto remoto sceglie la propria semantica (es. oggetto *unicast*, *multicast*, ..)
- Definisce il protocollo specifico con cui effettuare le chiamate remote
  - invocazioni *unicast* "*point-to-point*"
  - invocazioni verso gruppi di oggetti
  - strategie di riconnessione verso l'oggetto remoto
  - supporto alle strategie di replicazione di oggetti remoti
  - supporto per le strategie di attivazione degli oggetti remoti

Architettura di RMI: Localizzazione degli oggetti remoti



- I vari client vedono solamente l'interfaccia remota, ma non l'implementazione del servizio
- Quale meccanismo offre RMI per l'individuazione degli oggetti remoti?

Architettura di RMI: RMIRegistry

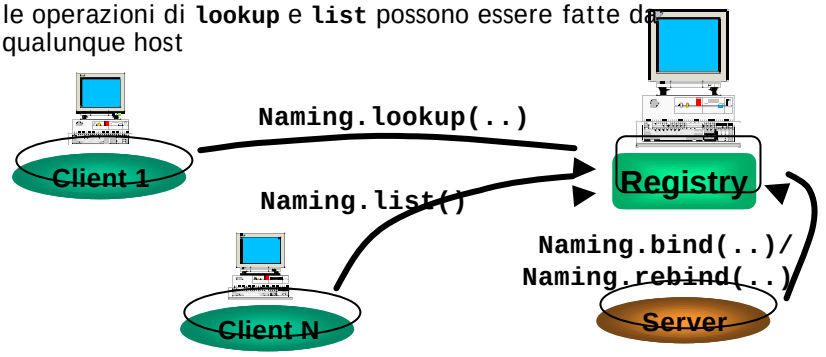
- Affinché un'applicazione client possa invocare metodi di un oggetto remoto, deve inizialmente ottenere un riferimento a tale oggetto (in seguito, può ottenere nuovi riferimenti come risultati di invocazione di metodi)
- JavaRMI fornisce un semplice *Name Server, chiamato* RMIRegistry, che viene utilizzato dal client in fase di bootstrap per recuperare il riferimento al "primo" oggetto remoto
- RMIRegistry è un processo che deve essere lanciato sulle macchine da cui si vogliono esportare oggetti remoti
- Il *Registry* è un oggetto che realizza una mappa fra nomi (stringhe) ed identificatori di oggetti remoti

Architettura di RMI: RMIRegistry (2)

- Operazioni tipiche effettuate sul registry dagli oggetti client e server:
  - **bind, rebind, unbind** (Associazione di un nome simbolico al riferimento ad un oggetto remoto, rimozione dell'associazione)
  - **lookup, list** (Richiesta del riferimento all'oggetto remoto associato ad un dato identificatore simbolico, lista dei nomi presenti)
- Queste operazioni sono implementate da metodi statici nella classe java.rmi.Naming

Architettura di RMI: RMIRegistry (3)

- Le operazioni di **`bind`**, **`rebind`**, **`unbind`** possono essere fatte solo dall'host su cui si trova il registro
  - per motivi di sicurezza: un client non deve modificare il registry del server
- le operazioni di **`lookup`** e **`list`** possono essere fatte da qualunque host



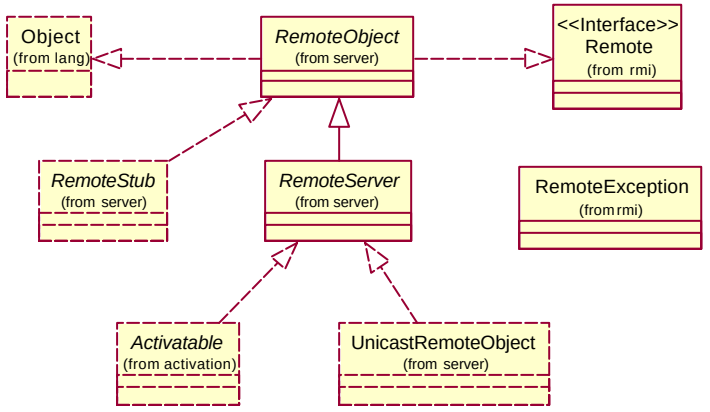
Il modello RMI: interfacce, eccezioni e classi

- Tutte le interfacce remote sono interfacce Java che devono estendere, direttamente o indirettamente, l'interfaccia java.rmi.Remote, la quale non definisce alcun metodo
- La classe java.rmi.RemoteException è la superclasse di tutte le eccezioni che possono essere sollevate dal sistema *runtime* di RMI
  - Per assicurare la robustezza di un'applicazione RMI, tutti i metodi dichiarati nelle interfacce remote devono dichiarare questa eccezione nella propria intestazione
  - Questa eccezione viene sollevata dal runtime di RMI quando l'invocazione di un metodo remoto fallisce (per esempio cade la connessione, o il server non può essere contattato, ...). Catturando tale eccezione nelle applicazioni client, è possibile svolgere delle azioni correttive

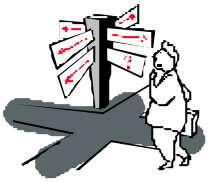
Il modello RMI: interfacce, eccezioni e classi

- Le funzioni di un server JavaRMI sono fornite dalla classe java.rmi.server.RemoteObject e dalle sue sottoclassi:
  - java.rmi.server.RemoteServer
  - java.rmi.server.UnicastRemoteObject
- RemoteObject fornisce la semantica remota degli oggetti, implementando i metodi *hashCode*, *equals*, *toString*
- Le funzioni necessarie a creare gli oggetti ed esportarli (rendendoli disponibili da remoto) sono fornite dalla classe RemoteServer e dalle sue sottoclassi.
  - A seconda della sottoclasse scelta, si possono avere diverse semantiche degli oggetti remoti esportati (oggetti singoli, replicati su più server, ...)
- La classe UnicastRemoteObject definisce il riferimento ad un unico (*Unicast*) oggetto remoto, il cui riferimento è valido solo se il processo server è attivo. Tale classe fornisce il supporto per gestire il riferimento all'oggetto

Il modello RMI: interfacce, eccezioni e classi



Indice degli argomenti



I punti essenziali per lo sviluppo di una semplice applicazione

- **Definizione dell'interfaccia remota**
  - Deve estendere `java.rmi.Remote`
  - Ciascun metodo deve dichiarare `java.rmi.RemoteException` nella lista delle possibili eccezioni sollevate
- **Definizione del codice del client**
  - richiede al registry un riferimento all'oggetto remoto
  - lo assegna ad una variabile che ha l'interfaccia remota come tipo
- **Definizione del codice dell'oggetto remoto**
  - deve implementare l'interfaccia remota
  - deve estendere la classe `java.rmi.server.UnicastRemoteObject`
- Creazione di stub e skeleton
  - si usa il comando rmic
- Definizione del codice del server
  - istanzia l'oggetto remoto e lo registra presso il registry

Esempio: l'interfaccia remota

```
import java.rmi.*;
public interface ProvaRMIServer
                    extends Remote {
  public void print(String s)
                        throws RemoteException;
  public int getNumber()
                        throws RemoteException;
}
```

- Ciascun metodo presente nell'interfaccia remota, oltre a dover sollevare *RemoteException*, può anche sollevare eccezioni specifiche dell'applicazione
- L'interfaccia remota deve essere dichiarata *public* per poter essere utilizzata da altre JavaVM. In caso contrario, un client otterrà un errore quando tenterà di ottenere un riferimento a tale oggetto

Naming Conventions

Suffisso	Descrizione
Nessun suffisso (es. Product)	Interfaccia remota
Suffisso Impl (es. ProductImpl)	Classe Server che implementa una interfaccia remota
Suffisso Client (es. ProductClient)	Client che invoca metodi sull'oggetto remoto
Suffisso _Stub (es. ProductImpl_Stub)	Classe stub automaticamente generata con rmic
Suffisso _Skel (es. ProductImpl_Skel)	Classe skeleton automaticamente generata con rmic (necessaria per JDK 1.1)

L'Implementazione di un'interfaccia remota

- Una classe remota:
  - Deve implementare l'interfaccia remota
  - Deve essere esportata per poter accettare richieste di servizio dai client
    - Può estendere (direttamente o indirettamente) la classe *UnicastRemoteObject*, da cui eredita il comportamento remoto fornito dalle classi *RemoteObject* e *RemoteServer*
      - In particolare l'esportazione automatica al momento della creazione (prevista nel costruttore vuoto)
    - Se non estende UnicastRemoteObject l'esportazione deve essere esplicitamente richiesta subito dopo la creazione (utilizzando il metodo statico exportObject di UnicastRemoteObject

L'Implementazione di un'interfaccia remota

- Un'unica classe può implementare più interfacce remote
- La classe può estendere un'altra implementazione di classe remota
- La classe può definire altri metodi oltre quelli presenti nell'interfaccia, ma questi metodi non saranno resi visibili all'esterno, per cui non potranno essere invocati da altre JavaVM.

Esempio: l'implementazione del server

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

public class ProvaRMIServerImpl extends
  UnicastRemoteObject implements ProvaRMIServer {
   int counter;
   public ProvaRMIServerImpl()throws
  RemoteException {}
   public void print(String s)throws
  RemoteException
     {System.out.println(s);}
   public int getNumber()throws RemoteException
     {return counter++; }
}
```


Esempio: l'implementazione del server

```
public static void main(String[] args) {
 try {
    System.setSecurityManager( new
            RMISecurityManager());
     ProvaRMIServerImpl server =
            new ProvaRMIServerImpl();
  Naming.rebind("//brahms/ProvaRMIServer",
            server);
  System.out.println("Server bound");
 } catch (Exception e) {
    e.printStackTrace();
 }
}
```

Non sempre strettamente necessario sul server

Formato generale: rmi://hostname:port/nomesimbolico

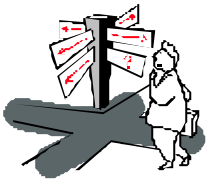
Esempio: il client

```
import java.rmi.*;
public class ProvaRMIClient {
  public static void main(String[] args) {
    try {
      System.setSecurityManager(
                    new RMISecurityManager());
       System.out.println("Looking up server...");
      ProvaRMIServer server = (ProvaRMIServer)
       Naming.lookup("//"+args[0]+"/ProvaRMIServer
       ");
      System.out.println("Server bound...");
      server.print("prima prova");
      System.out.println(server.getNumber());
    } catch (Exception e) {
      e.printStackTrace();
    }
  } }
```

Esempio: la compilazione

- `javac ProvaRMIServerImpl.java`
  - compila il server
- `javac ProvaRMIClient.java`
  - compila il client
- `rmic ProvaRMIServerImpl`
  - crea le classi `ProvaRMIServer_Skel` e `ProvaRMIServer_Stub`

Indice degli argomenti



Esecuzione di una applicazione

- `rmiregistry`
  - lancia il registry
- `java ProvaRMIServerImpl`
  - lancia il server
- `java ProvaRMIClient localhost`
  - lancia il client dicendo di collegarsi a localhost
- Con queste impostazioni il tutto funziona solo se
  - Lo stub dell'oggetto remoto è nel classpath del client
  - Lo stub dell'oggetto remoto è "visibile" per il registry (attraverso il classpath impostato al momento del lancio)
  - Non viene impostato un security manager nel client (e nel server)
- In realtà in questo modo non è stata realizzata un'applicazione realmente distribuita (o meglio, realmente distribuibile)!!
  - Per raggiungere lo scopo occorrono ulteriori impostazioni

Caricamento dinamico delle classi

- Al contrario di altre architetture per sistemi distribuiti, in cui gli stub devono essere sempre visibili al client, RMI può caricare dinamicamente
  - Gli stub degli oggetti remoti
  - Tutte le altre classi coinvolte nell'invocazione (es. parametri e valori di ritorno) non disponibili nel classpath del chiamante
- Per caricare gli oggetti da siti remoti, RMI usa l'*Object Serialization*. Quindi tutte le classi caricate devono essere serializzabili

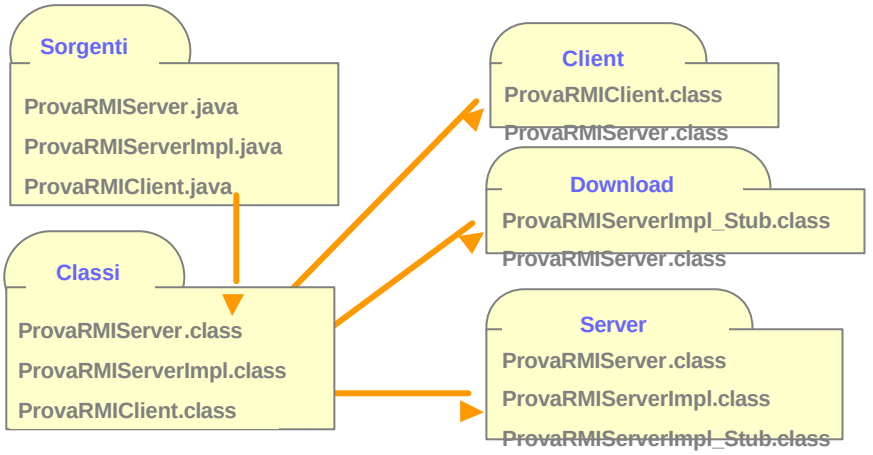
Caricamento dinamico delle classi

- Le classi possono essere scaricate attraverso un web server
- Per indicare dove trovare gli stubs che dovranno essere scaricati dinamicamente, si lancia il server con il seguente comando:

```
java -Djava.rmi.server.codebase=
"http://brahms:8000/" ProvaRMIServerImpl
```

- Perchè client e server possano istanziare correttamente le classi scaricate, è necessario configurare il *security manager*
  - Questo impedisce alle classi scaricate di eseguire azioni non legali

Posizionamento dei file nelle directory



Caricamento dinamico delle classi: Osservazioni

- Gli URL specificati con la proprietà java.rmi.server.codebase
  - devono consentire l'accesso agli stub degli oggetti remoti per il caricamento dinamico ad opera del client
  - devono conesntire l'accesso anche a TUTTE le classi dalle quali gli stub dipendono
    - in particolare anche l'interfaccia remota deve essere raggiungibile attraverso gli URL specificati con java.rmi.server.codebase (e quindi essere contenuta nella directory download)
    - se questo vincolo non viene rispettato l'operazione di bind o rebind effettuata dal server sull'rmiregistry non va a buon fine
- La classe stub del server remoto deve sempre essere contenuta nella directory "server", pena il fallimento delle operazioni di bind, rebind effettuate dal server

Sicurezza in RMI

- Le classi prelevate da remoto sono considerate "non affidabili"
  - E' necessario istanziare un opportuno SecurityManager
- Se non è impostato un *SecurityManager*, il caricamento delle classi remote è disabilitato (ogni tentativo genera un errore)
- Il *SecurityManager* deve essere installato come prima operazione
- Gli *applet* sono soggetti alle restrizioni del security manager attivo nella JVM del browser (controlli sul codice+host di provenienza)

Sicurezza in RMI (2)

- Un security manager richiede l'utilizzo di un file di policies nel quale vengono specificati i permessi di effettuare diverse tipologie di operazioni da parte delle classi caricate dai diversi codebase
  - Il file può essere collocato in una qualsiasi directory e può avere un nome qualsiasi
  - La posizione del file di policy viene specificata attraverso la proprietà java.security.policy sulla riga di comando

```
java –
  Djava.security.policy=c:\<dir>\<nomefile>
  ProvaRMIClient localhost
```


Sicurezza in RMI (3)

- Per semplicità si può impostare un file di policy in cui viene definito una "global permission"
  - chiunque può fare qualsiasi cosa
- Il policy file "policy.java" può essere creato utilizzando un normale text editor oppure attraverso il tool "policytool"

```
grant {
        // Allow everything for now
        permission java.security.AllPermission;
};
```

- Ovviamente non è opportuno utilizzare una policy di questo tipo in un production environment….

Sicurezza in RMI (4)

- Occorre impostare le policies
  - sul client
  - sul server se nel codice viene impostato un security manager
- Client e server dovranno essere eseguiti impostando la proprietà java.security.policy con il path del policy file creato:
  - Sul server

    java -Djava.security.policy=.\policy.java ProvaRMIServerImpl
  - Sul client

    java -Djava.security.policy=.\policy.java ProvaRMIClient

Sicurezza in RMI (5)

- Formato generale di una entry in un file di policy

```
grant [codeBase "URL"] { permission
  permissionClassName "target", "action";
  //... };
```

  - Definisce in modo dettagliato i permessi attribuiti alle classi caricate dal codebase specificato
    - Se non viene specificato il codebase le permission si applicano a tutte le classi, indipendentemente dalla loro provenienza

Sicurezza in RMI (6)

- Un esempio di policy file più dettagliato:

```
grant {
  permission java.net.SocketPermission
  "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission "*:80",
  "connect"; };
```

  - Possibilità di connettersi a qualsiasi host e di accettare connessioni da qualsiasi host e su qualsiasi porta da 1024 a 65535 (utile per connettersi a rmiregistry)
  - Possibilità di connettersi a qualsiasi host sulla porta 80 (http server, ad esempio per il download dinamico delle classi)

Sicurezza in RMI (7)

- Informazioni su policy files e permissions:
  - Default Policy Implementation and Policy File Syntax:
    - http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html
  - Permissions in JDK1.2:
    - http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html

Esecuzione dell'applicazione

- Attivazione del registry (classpath vuoto)

```
rmiregistry
```

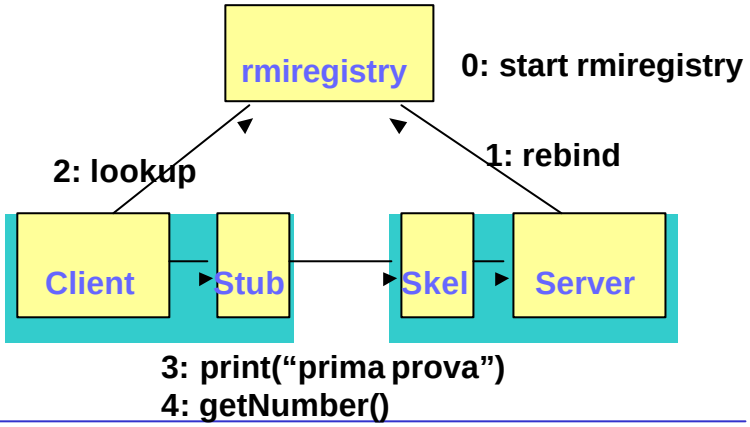
- Esecuzione del server (classpath contenente la directory server)

```
java
-Djava.security.policy=.\policy.java
-Djava.rmi.server.codebase=
"http://brahms:8000/.../download" ProvaRMIServerImpl
```

- Esecuzione del client (classpath contenente la directory client)

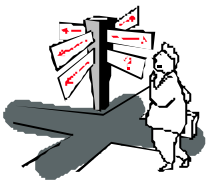
```
java
-Djava.security.policy=.\policy.java  ProvaRMIClient
    localhost
```


Esempio: Le interazioni tra i vari componenti



Indice degli argomenti

- Contesto e motivazioni
- Architettura e concetti di base
  - Componenti di RMI
  - RMIRegistry
  - Interfacce, eccezioni e classi
- Lo sviluppo di una applicazione
- L'esecuzione di una applicazione
  - Caricamento dinamico delle classi
  - RMI e sicurezza
- Il passaggio dei parametri
- L'utilizzo del registry



Argomenti e risultati di metodi remoti

- Un argomento o valore di ritorno di un oggetto remoto può essere qualunque tipo Java che sia *Serializable*
  - tipi Java primitivi
  - oggetti Java remoti
  - oggetti Java (anche non remoti) che implementano l'interfaccia *java.io.Serializable*
- Per gli *applet*, se la definizione della classe di un parametro o risultato non è disponibile localmente, viene caricata dinamicamente dall'*AppletClassLoader*
- Per le applicazioni, se la definizione della classe di un parametro (o risultato) non è stata ancora caricata, viene caricata o dal *ClassLoader* locale (che fa riferimento al CLASSPATH locale), o dall'*RMIClassLoader*, che usa la proprietà *java.rmi.server.codebase* del server

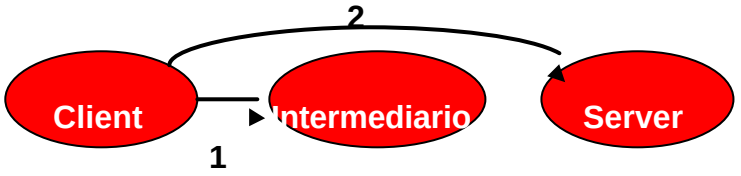
Il passaggio dei parametri ed i valori di ritorno

- Due tipologie di parametri e valori di ritorno:
  - Riferimenti ad oggetti remoti
    - Si acquisisce un riferimento all'oggetto.
    - L'oggetto rimane unico e risiede sul server, anche se più componenti ne acquisiscono il riferimento
      - viene passato per copia lo stub dell'oggetto
    - Un oggetto remoto passato come parametro può implementare solo interfacce remote e non interfacce locali
  - Riferimenti ad oggetti non remoti
    - Una copia dell'oggetto viene trasferita sul sito remoto
      - se un parametro non remoto passato da client al server viene da quest'ultimo modificato, il client non risente di tale modifica
    - l'oggetto locale deve essere *serializzabile*

Passaggio di riferimenti ad oggetti remoti

```
import java.rmi.*;

public interface ProvaRMIIntermed extends
  Remote {
  public ProvaRMIServer getProvaRMISrv()
  throws RemoteException;
}
```



Passaggio di riferimenti ad oggetti remoti: Implementazione del server intermedio

```
public class ProvaRMIIntermedImpl extends
  UnicastRemoteObject implements ProvaRMIIntermed
  {
  ProvaRMIServer s;
  public ProvaRMIServer getProvaRMISrv() throws
  RemoteException { return s; }
  public static void main(String[] args) {
    try {
      System.setSecurityManager(new
            RMISecurityManager());
      System.out.println("Looking up server...");
      s = (ProvaRMIServer) Naming.lookup("//" +
            args[0] + "/" args[1]);
      System.out.println("Server located...");
  }
}
```


Passaggio di riferimenti ad oggetti remoti: Implementazione del client

```
import java.rmi.*;

public class ProvaRMIInterClient {
  public static void main(String[] args) {
    try {
      System.setSecurityManager(new
  RMISecurityManager());
      System.out.println("Looking up server...");
      ProvaRMIIntermed s1 = (ProvaRMIIntermed)
          Naming.lookup(args[0] + args[1]);
      System.out.println("Intermed located...");
```

Passaggio di riferimenti ad oggetti remoti: Implementazione del client (2)

```
      ProvaRMIServer server =
            s1.getProvaRMISrv();
      System.out.println("Server located...");
      server.print("prima prova");
      System.out.println(server.getNumber());
    } catch (Exception e) {
      e.printStackTrace();
    }
 }
}
```


Passaggio di riferimenti ad oggetti non remoti

```
import java.rmi.*;

public interface RMIServer extends Remote {
  public void print(String s) throws
   RemoteException;
  public Persona getOwner() throws
   RemoteException;
}
```

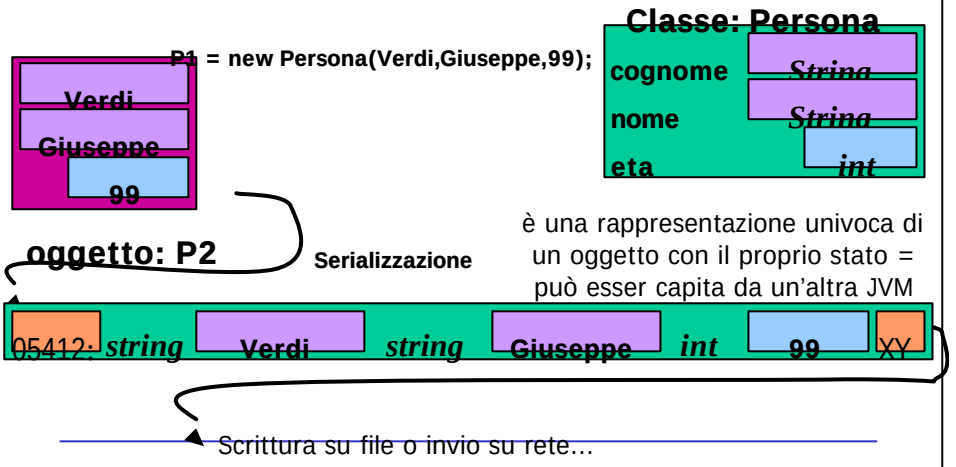

Riferimenti ad oggetti non remoti

```
public class Persona implements
  java.io.Serializable {
  protected String nome=null;
  protected String cognome=null;
  protected int eta=0;

  public Persona(String _nome, String _cognome,
  int _eta) {
  nome=_nome; cognome=_cognome; eta=_eta; }
  public void parla() {
    System.out.println("Sono una persona di
  "+eta+" anni");
  }
}
```


Serializzazione: cenni

- Da rappresentazione in memoria a versione serializzata



Passaggio di parametri e caricamento dinamico delle classi

- Uno dei principi di java è quello dello “zero deployment”
- Quando si usa RMI è possibile scaricare dal sito del server:
  - Gli stub degli oggetti remoti
  - Le classi corrispondenti ai parametri
    - Per esempio, il client riceve un valore di ritorno da una chiamata remota che ha un certo tipo statico noto al client, ed un tipo dinamico non noto
- Lo scaricamento è possibile anche dal client verso il server
    - Ad esempio quando il server riceve un parametro che ha un tipo statico noto e un tipo dinamico non noto
    - In questo caso occorre impostare la proprietà java.rmi.server.codebase anche sul client

Esempio

- Sul ***client***

```
public class Uomo extends Persona {
public Uomo(String _nome, String _cognome, int _eta) {
   super(_nome, _cognome, _eta);
}
public void parla() {
   System.out.println("Sono un uomo di "+eta+" anni");
}}
```

- Sul ***server***

```
public class Bambino extends Persona {
public Bambino(String _nome, String _cognome, int _eta) {
   super(_nome, _cognome, _eta);
}
public void parla() {
 System.out.println("Sono un bambino di "+eta+" anni");
}}
```

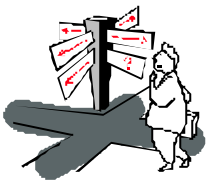

Esempio: Implementazione del server

```
public class RMIServerImpl extends UnicastRemoteObject
    implements RMIServer {
      Persona persona = null;
       ....
      public void setOwner(Persona p) throws RemoteException {
          persona = p;
         return;
      }
      public Persona getOwner() throws RemoteException {
        Persona current = persona;
        persona = new Bambino("Giuseppe", "Garibaldi", 2);
        return current;
      }
  }
```


Esempio: Implementazione del client

```
public class ProvaRMIClient {
  public static void main(String[] args) {
    try {
      System.setSecurityManager( new RMISecurityManager());
      System.out.println("Looking up server...");
      ProvaRMIServer server = (ProvaRMIServer)
     Naming.lookup("//"+args[0]+ "/ProvaRMIServer");
      System.out.println(server.getNumber());
      Uomo u = new Uomo("Giuseppe", "Verdi", 99);
      server.setOwner(u);
      Persona p = server.getOwner(); p.parla();
      p = server.getOwner(); p.parla();
    } catch (Exception e) {
      e.printStackTrace();
    } } }
```

Indice degli argomenti



Registrazione di oggetti remoti sul Registry: Approfondimenti

- JavaRMI fornisce un semplice *Name Server* tramite i cinque metodi statici della classe *java.rmi.Naming*
- Il *Name Server* di RMI è basato sull'*Uniform Resource Locator* (URL) del server dove si trovano gli oggetti remoti
  - Non fornisce trasparenza alla locazione degli oggetti
- Il *Name Server* è un processo (*Registry*) che deve essere lanciato sulle macchine da cui si vogliono esportare oggetti remoti
- Il *Registry* è un oggetto che realizza una mappa fra nomi (stringhe) ed identificatori di oggetti remoti

Registrazione di oggetti remoti sul Registry

- Quando il *Name Server* è stato attivato, tutte le applicazioni Java che intendono esportare oggetti remoti devono registrare tali oggetti presso il *Registry*, assegnando a ciascun oggetto un nome (operazione di *bind*)
- Si può scegliere il numero di porta su cui lanciare il Registry (1099 di default), cosicché ogni processo server può utilizzare un proprio *Registry*, oppure un singolo registro può essere utilizzato per tutti gli oggetti su un host
- E' possibile utilizzare la classe java.rmi.registry.LocateRegistry e l'interfaccia java.rmi.registry.Registry per accedere da programma ad un registry o per crearne uno nuovo

La classe LocateRegistry

```
public final class LocateRegistry {
    public static Registry getRegistry();
    // permette di ottenere un riferimento al registro sull'host corrente che è in attesa sulla porta di default (1099)
    public static Registry getRegistry(int port);
    // permette di ottenere un riferimento al registro sull'host corrente che è in attesa sulla porta passata come argomento
    public static Registry getRegistry(String host);
    // permette di ottenere un riferimento al registro sull'host passato come argomento che è in attesa sulla porta di default (1099)
    public static Registry getRegistry(String host, int port);
    // permette di ottenere un riferimento al registro sull'host passato come primo argomento che è in attesa sulla porta passata come secondo argomento
    public static Registry createRegistry(int port);
    // permette di creare un registro sull'host corrente e sulla porta specificata
}
```


L'interfaccia del registry

```
public interface Registry extends java.rmi.Remote {
    public java.rmi.Remote lookup(String name);
    // restituisce l'identificatore dell'oggetto remoto legato al nome specificato
    public void bind(String name, java.rmi.Remote obj);
    // memorizza nel registro il riferimento all'oggetto remoto passato come
    secondo parametro con un nome dato dal primo parametro
    public void rebind(String name, java.rmi.Remote obj);
    // stessa operazione di bind: unica differenza risiede nel fatto che se un
    oggetto con lo stesso nome era già presente nel registro, viene scaricato
    public void unbind(String name);
    // rimuove il riferimento dell'oggetto con il nome specificato dal registro
    public String[] list();
    // restituisce al chiamante la lista dei nomi degli oggetti presenti nel registro
}
```


Bibliografia

- G. Cornell, C. S. Horstmann. *Core Java 2*. SunSoft Press, Prentice Hall. 2nd edition, 1999.
- *Java RMI Tutorial*. Sun Microsystems, Inc. http://www.javasoft.com
- *Java Remote Method Invocation Specification*. Sun Microsystems, Inc., 1999. http://www.javasoft.com

