



**Università degli Studi di Bologna
Facoltà di Ingegneria**

Corso di
Reti di Calcolatori L-A
Java RMI
(Remote Method Invocation)

Luca Foschini

Anno accademico 2009/2010

RMI: motivazioni e generalità

RPC in JAVA: le RMI introducono la possibilità di **richiedere esecuzione di metodi remoti in JAVA** integrando il tutto con il **paradigma OO**

Definizioni e generalità

Insieme di **strumenti, politiche e meccanismi** che permettono ad un'applicazione Java in esecuzione su una macchina di **invocare i metodi di un oggetto di una applicazione Java in esecuzione su una macchina remota**

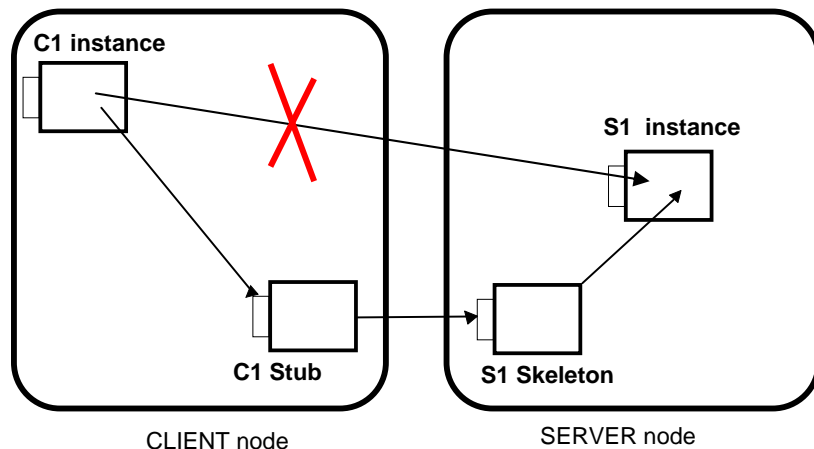
Viene creato localmente solo il **riferimento ad un oggetto remoto**, che è invece effettivamente attivo su un nodo remoto

Un programma cliente invoca i metodi attraverso questo **riferimento locale**

Unico ambiente di lavoro come conseguenza del linguaggio Java, ma **Eterogeneità di sistemi** → grazie a **portabilità codice Java (BYTECODE)**

Accesso ad oggetti remoti

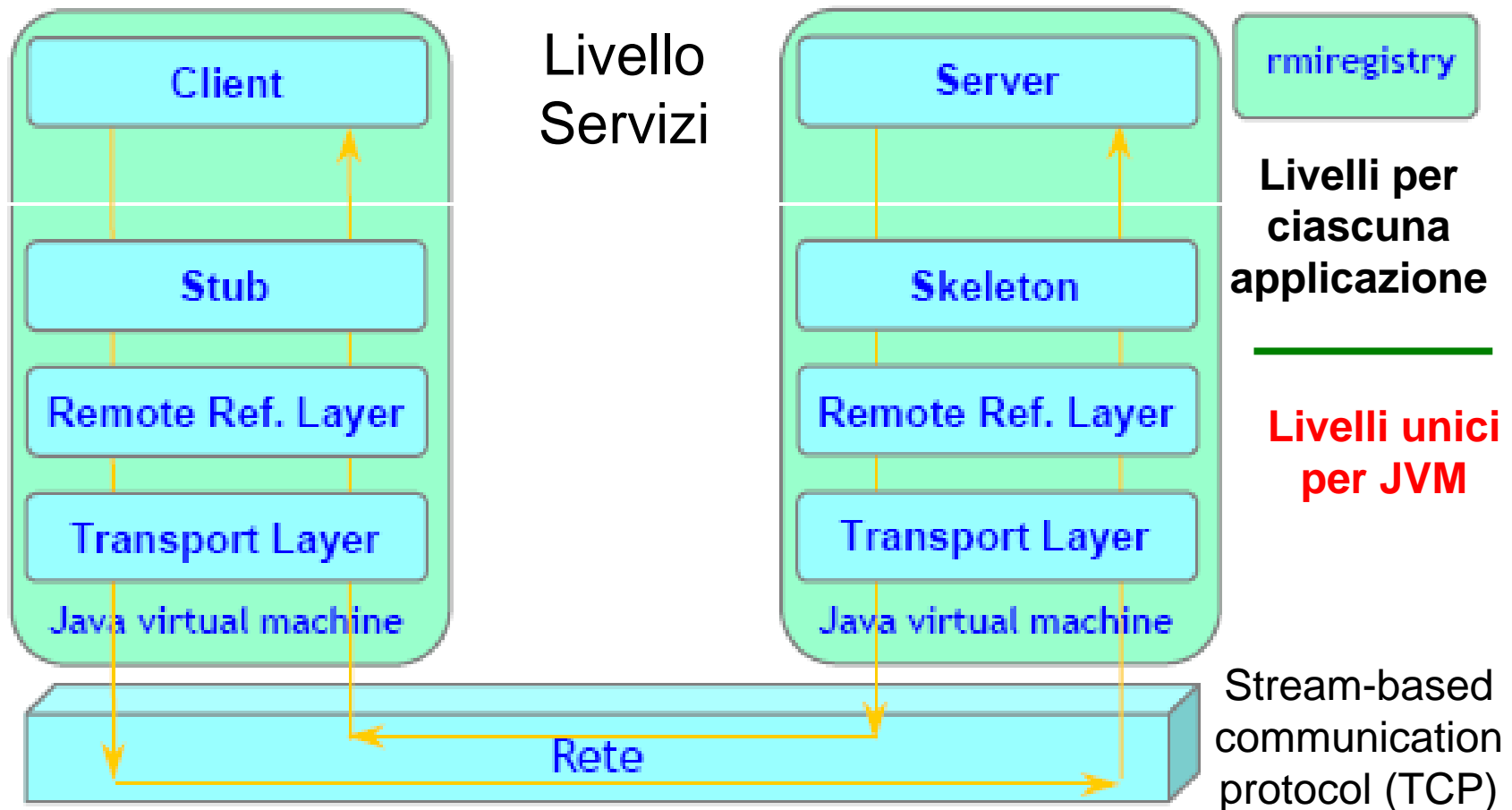
- In Java **non** sono **(direttamente) disponibili riferimenti remoti**, ma si possono costruire con RMI
- Remote Method Invocation
 - Due **proxy**: **stub** dalla parte cliente e **skeleton** dalla parte servitore
 - Proxy pattern: questi componenti **nascondono** al livello applicativo **la natura distribuita dell'applicazione**



- Cosa cambia rispetto ad una invocazione di oggetto locale?
 - Affidabilità, semantica, durata...
- **NOTA:** **non** è possibile riferire **direttamente** l'oggetto remoto → **necessità di una infrastruttura attiva e distribuita**

Architettura RMI

Solo interazioni **SINCRONE** e **BLOCCANTI**



Architettura RMI a livelli

- **Stub e skeleton:**
 - **Stub**: **proxy locale** su cui vengono fatte le invocazioni destinate all'oggetto remoto
 - **Skeleton**: **entità remota** che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sul server
- **Lo strato Remote Reference Layer (RRL):**
 - Responsabile della gestione dei riferimenti agli oggetti remoti, dei parametri e delle astrazioni di stream-oriented connection
- **Il Transport Layer**
 - Responsabile della gestione delle connessioni fra i diversi address space (JVM diverse)
 - Gestisce il ciclo di vita delle connessioni e le attivazioni integrate in JVM
 - Può utilizzare protocolli applicativi diversi, purché siano connection-oriented → **TCP a livello di trasporto**
 - Utilizza un protocollo proprietario
- **Registry**: **servizio di nomi** che consente al server di pubblicare un servizio e al client di recuperarne il proxy

Caratteristiche RMI

Modello a oggetti distribuito

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

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

Chiamata locale vs. chiamata remota

Il cliente invoca un metodo di un oggetto non locale

- **Sintassi:** uguale → trasparenza
- Chiamata **sincrona** e **bloccante** sempre con attesa
- **Semantica:** diversa
 - Chiamate locali → affidabilità massima
 - Chiamate remote: comunicazione con possibilità di fallimento
→ **semantica “at most once”** con uso TCP
 - Server remoto come locale: ogni chiamata esegue in modo **indipendente** e **parallelo (?)**

Interfacce e Implementazione

- Separazione tra
 - Definizione del comportamento → **interfacce**
 - Implementazione del comportamento → **classi**
- Realizzare componenti remoti
 1. **Definizione** del comportamento, **interfaccia** che
 - Estende `java.rmi.Remote`
 - Propaga `java.rmi.RemoteException`
 2. **Implementazione** comportamento, **classe** che
 - Implementa l'interfaccia definita
 - Estende `java.rmi.UnicastRemoteObject`

Passi per l'utilizzo di Java RMI

È necessario:

1. **Definire** **interfacce** e **implementazioni** dei componenti utilizzabili in remoto
2. **Compilare** le classi (con javac) e **generare** **stub** e **skeleton** (con rmic) delle classi utilizzabili in remoto
3. **Pubblicare** il servizio nel sistema di nomi registry
 - attivare il **registry**
 - registrare il **servizio** (il server deve fare una bind sul registry)
4. **Ottenere** (lato client) il riferimento all'oggetto remoto tramite il name service, facendo una **lookup sul registry**

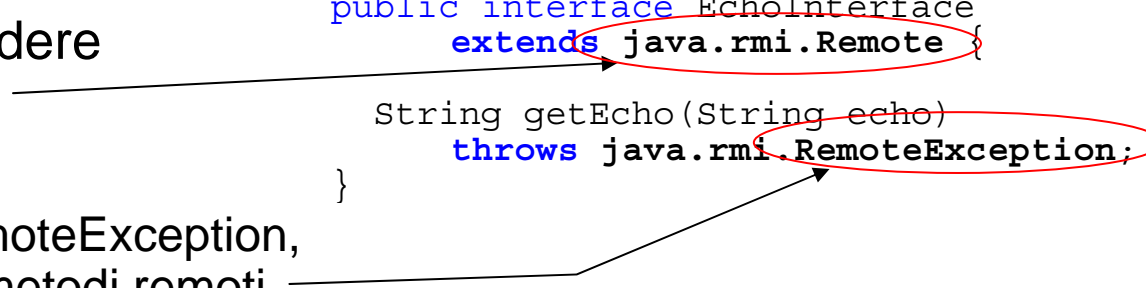
A questo punto l'interazione tra il cliente e il server può procedere

N.B.: questa è una descrizione di base, dettagli sul registry e sul caricamento dinamico delle classi saranno dati in seguito.

Implementazione: Interfaccia

- L'interfaccia deve estendere l'interfaccia Remote
- Ciascun metodo remoto
 - Deve lanciare una RemoteException, cioè l'invocazione dei metodi remoti **NON** è completamente trasparente
 - Ha **un solo** parametro di uscita e **nessuno, uno o più** parametri di ingresso
 - I parametri **devono** essere passati **per valore** (dati **primitivi** o oggetti **Serializable**) o **per riferimento** (oggetti **Remote**) → l'ultimo aspetto sarà ripreso quando parleremo del passaggio dei parametri

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



Implementazione: Server

La classe che implementa il server

- Estende la classe UnicastRemoteObject
- Implementa **tutti i metodi definiti nell'interfaccia**

```
public class EchoRMIServer
    extends java.rmi.server.UnicastRemoteObject
    implements EchoInterface{
```

```
// Costruttore
public EchoRMIServer()
    throws java.rmi.RemoteException
{ super(); }
```

```
// Implementazione del metodo remoto
    dichiarato nell'interfaccia
public String getEcho(String echo)
    throws java.rmi.RemoteException
{ return echo; }
```

Un processo in esecuzione sull'host del servitore registra tutti i servizi

- Invoca tante bind/rebind quanti sono gli **oggetti server** da registrare ciascuno **con un nome logico**

```
public static void main(String[] args){
    // Registrazione del servizio
    try
    {
        EchoRMIServer serverRMI =
            new EchoRMIServer();
        Naming.rebind("EchoService", serverRMI);
    }
    catch (Exception e)
    { e.printStackTrace(); System.exit(1); }
}
```

Registrazione del servizio

- Bind e rebind possibili **solo** sul registry locale

Implementazione: Client

Servizi acceduti attraverso la **variabile interfaccia** ottenuta **con una richiesta al registry**

Reperimento di **un riferimento remoto** cioè una **istanza di stub** dell'oggetto remoto (**NON** della **classe** dello stub, che solitamente si assume già presente sul client)

Invocazione metodo remoto

- Chiamata **sincrona bloccante** con i parametri specificati in interfaccia

```
public class EchoRMIClient
{
    // Avvio del Client RMI
    public static void main(String[] args)
    {
        BufferedReader stdIn=
            new BufferedReader(
                new InputStreamReader(System.in));

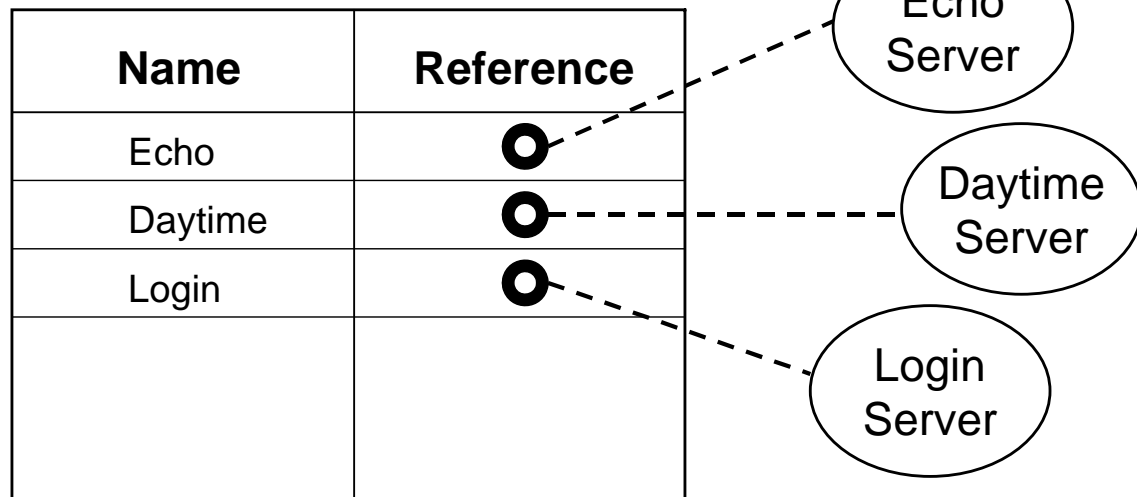
        try
        {
            // Connessione al servizio RMI remoto
            EchoInterface serverRMI = (EchoInterface)
                java.rmi.Naming.lookup("EchoService");

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

            // Richiesta del servizio remoto
            echo = serverRMI.getEcho(message);
            System.out.println("Echo: "+echo+"\n");
        }
        catch (Exception e)
        { e.printStackTrace(); System.exit(1); }
    }
}
```

RMI Registry

- **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 indirizzione**
- Java RMI utilizza un naming service: **RMI Registry**
- Mantiene un insieme di coppie **{name, reference}**
 - Name: stringa arbitraria non interpretata
- **NON** c'è trasparenza alla locazione



Classe Naming e Attivazione Registry

Metodi della classe java.rmi.Naming:

```
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)
```

Ognuno di questi metodi crea una connessione (con una socket) con il registry identificato da host e porta

name -> combina la locazione del registry e il nome logico del servizio, nel formato: `//registryHost:port/logical_name`

- `registryHost` = macchina su cui eseguono il registry e i servitori
- `port` = 1099 a default
- `logical_name` = il nome del servizio che vogliamo accedere

**Non c'è
trasparenza
alla
locazione!!**

Attivazione registry (sull'host del server): usare **programma rmiregistry**

di Sun lanciato in una shell a parte specificando o meno la porta (default 1099):

`rmiregistry` oppure `rmiregistry 10345`

N.B.: il registry è attivato così in una istanza separata della JVM

Compilazione e Esecuzione

Lato server

1. Compilazione **interfaccia e implementazione** parte server

```
javac      EchoInterface.java  
          EchoRMIServer.java
```

2. Generazione **eseguibili Stub e Skeleton**

```
rmic [-vcompat] EchoRMIServer
```

→

```
EchoRMIServer_Stub.class  
EchoRMIServer_Skel.class
```

Nota: in Java 1.5 e seguenti invocare **rmic** con opzione **-vcompat**

3. Esecuzione lato server (registry e server)

- Avviamento del registry: **rmiregistry**
- Avviamento del server: **java EchoRMIServer**

Lato client

1. Compilazione: **javac EchoRMIClient.java**
2. Esecuzione: **java EchoRMIClient**

Passaggio dei parametri

Tipo	Metodo Locale	Metodo Remoto
Tipi primitivi	Per valore	Per valore
Oggetti	Per riferimento	Per valore (interfaccia Serializable e deep copy)
Oggetti Remoti		Per riferimento remoto (interfaccia Remote)

In **Locale**:

- **Copia** → **tipi primitivi**
- **Per riferimento** → **tutti gli oggetti Java (tramite indirizzo)**

In **Remoto**: **(problemi nel riferire entità non locali)**

- **Passaggio per valore** → **tipi primitivi e Serializable Object**
 - 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
- **Passaggio per riferimento remoto** → **Remote Object** **via RMI**
 - Oggetti la cui funzione è **strettamente legata alla località in cui eseguono** (server) sono passati **per riferimento**: ne viene serializzato lo stub, creato automaticamente a partire dalla classe dello stub. Ogni istanza di stub identifica l'oggetto remoto al quale si riferisce attraverso un **identificativo** (ObjID) che è **univoco** rispetto alla JVM dove l'oggetto remoto si trova

La serializzazione

- **In generale**, in sistemi RPC i parametri di ingresso e uscita subiscono una duplice trasformazione per risolvere problemi di rappresentazioni eterogenee
 - **Marshalling**: processo di codifica degli argomenti e dei risultati per la trasmissione
 - **Unmarshalling**: processo inverso di decodifica di argomenti e risultati ricevuti
- **In Java**, grazie all'uso del BYTECODE, **NON** c'è bisogno di un/marshalling, ma i dati vengono semplicemente serializzati/deserializzati utilizzando le funzionalità offerte **direttamente a livello di linguaggio**
- **Serializzazione**: trasformazione di oggetti complessi in semplici sequenze di byte
 - metodo **writeObject()** su uno stream di output
- **Deserializzazione**: decodifica di una sequenza di byte e costruzione di una copia dell'oggetto originale
 - metodo **readObject()** da uno stream di input
- Stub e skeleton utilizzano queste due funzionalità per lo scambio dei parametri di ingresso e uscita con l'host remoto

Serializzazione: interazione con stream per TX/RX

Esempio di **oggetto serializzabile “Record”** con scrittura su stream

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

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

Si possono usare soltanto istanze di oggetti serializzabili, ovvero che:

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

NOTA BENE:

NON viene **trasferito l'oggetto vero** e proprio ma solo le informazioni contenute che caratterizzano l'istanza

- **no** metodi, **no** costanti, **no** variabili **static**, **no** variabili **transient**

Al momento della deserializzazione sarà **ricreata una copia** dell'istanza “trasmessa” usando il **.class** (che deve quindi essere accessibile!!!) dell'oggetto e le informazioni ricevute

Serializzazione: esempio

Riprendendo il server di echo

⇒ messaggio passato come **oggetto serializzabile** anziché come stringa

```
public class Message implements Serializable
{
    String content;
    // ... altri eventuali campi

    // Costruttore
    public Message(String msg){ content=msg; }

    public String toString(){ return content; }
}
```

L'oggetto viene trasferito come **contenuto completo**

Stub e Skeleton

- Stub e Skeleton
 - Rendono possibile la chiamata di un servizio remoto come se fosse locale (agiscono da proxy)
 - Generati dal **compilatore RMI**
 - De/serializzazione supportata **direttamente dall'ambiente di sviluppo Java**
- Procedura di comunicazione
 1. il client ottiene un'istanza dello stub
 2. il client chiama metodi sullo stub
 3. lo stub:
 - effettua la serializzazione delle informazioni per la chiamata (id del metodo e argomenti)
 - invia le informazioni allo skeleton utilizzando le astrazioni messe a disposizione dal RRL
 4. lo skeleton:
 - effettua la de-serializzazione dei dati ricevuti
 - invoca la chiamata sull'oggetto che implementa il server (dispatching)
 - effettua la serializzazione del valore di ritorno e invio allo stub
 5. lo stub:
 - effettua la de-serializzazione del valore di ritorno
 - restituisce il risultato al client

Implementazione del Registry

Il Registry **è un server RMI**

- Interfaccia: `java.rmi.registry.Registry`
- Classe d'implementazione: `sun.rmi.registry.RegistryImpl`

```
public interface Registry extends Remote {  
    public static final int REGISTRY_PORT = 1099;  
    public Remote lookup(String name)  
        throws RemoteException, NotBoundException, AccessException;  
    public void bind(String name, Remote obj)  
        throws RemoteException, AlreadyBoundException, AccessException;  
    public static void rebind(String name, Remote obj)  
        throws RemoteException, AccessException;  
    public static void unbind(String name)  
        throws RemoteException, NotBoundException, AccessException;  
    public static String[] list(String name)  
        throws RemoteException, AccessException;  
}
```

Creare e lanciare **all'interno del codice** (del server) un proprio **registry**:

```
public static Registry createRegistry(int port)
```

- È un metodo della classe `LocateRegistry`
- Il registry viene creato nella **stessa istanza** della JVM

Approfondimenti su RMI

Stub

- Si appoggia sul Remote Reference Layer (RRL)
 - Estende `java.rmi.server.RemoteStub`
 - Implementa `java.rmi.Remote` e l'interfaccia remota del server (es. `EchoInterface`)
 - Contiene al suo interno un'istanza del riferimento all'oggetto remoto (`super.ref` di classe `java.rmi.server.RemoteRef`)
- Lo stub **effettua l'invocazione, gestisce la de/serializzazione, e spedisce/riceve gli argomenti e il risultato**

Intero indicante

l'operazione richiesta

```
...
// creazione della chiamata
java.rmi.server.RemoteCall remotecall =
super.ref.newCall(this, operations,
    0, 6658547101130801417L);
// serializzazione dei parametri
try{
    ObjectOutputStream objectoutput =
        remotecall.getOutputStream();
    objectoutput.writeObject(message);
}
...
// invocazione della chiamata, sul RRL
super.ref.invoke(remotecall);
...
```

```
// de-serializzazione del valore di ritorno
String message1;
try{
    ObjectInput objectinput =
        remotecall.getInputStream();
    message1 = (String)objectinput.readObject();
}
...
// segnalazione chiamata andata a buon fine al RRL
finally{
    super.ref.done(remotecall); //a cosa serve!?!?
}
// restituzione del risultato
// al livello applicativo
return message1;
...
```

Skeleton

- Metodo **dispatch** invocato dal RRL, con parametri d'ingresso
 - Riferimento al server (`java.rmi.server.Remote`)
 - Chiamata remota, numero operazione, e hash dell'interfaccia
- Lo skeleton **gestisce** la **de/serializzazione**, **spedisce/riceve** i dati appoggiandosi sul RRL, ed **invoca il metodo** richiesto (**dispatching**)

```
public void dispatch(Remote remote,
    RemoteCall remotecall,
    int opnum, long hash)throws Exception{
    ...
    EchoRMIServer echormiserver =
        (EchoRMIServer)remote;
    switch(opnum){
        case 0: // operazione 0
            String message;
            try{ // de-serializzazione parametri
                ObjectInput objectinput =
                    remotecall.getInputStream();
                message =
                    (String)objectinput.readObject();
            }
            catch(...){...}
            finally{ // libera il canale di input
                remotecall.releaseInputStream();
            }
            // invocazione metodo
            String message1 = echormiserver.getEcho(message);
            try{ // serializzazione del valore di ritorno
                ObjectOutput objectoutput =
                    remotecall.getResultStream(true);
                objectoutput.writeObject(message1);
            }
            catch(...){...}
            break;
            ... // gestione di eventuali altri metodi
            default:
                throw new UnmarshalException("invalid ...");
            } //switch
        } // dispatch
    }
```

Livello di trasporto: la concorrenza

- Specifica molto **aperta** e **non completa**
 - **Comunicazione** e **concorrenza** sono **aspetti chiave**
 - **Libertà** di realizzare diverse implementazioni **ma**
- Implementazione → **Server parallelo**

*“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”*

Cioè al livello applicativo dobbiamo comunque tenere in conto **problematiche di sincronizzazione** →
uso di lock: **synchronized**

- Thread usage in RMI (dalla specifica) → **tipicamente generazione**

*“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 remote object invocations to threads”*

Questo implica di avere un thread per ogni invocazione sull'oggetto remoto in esecuzione sulla JVM
(dove è espressa la concorrenza e generato il thread?)

Livello di trasporto: la comunicazione

- Anche in questo caso la specifica è molto **aperta**
 - Stabilisce unicamente un principio di buon utilizzo delle risorse
 - Se esiste già una connessione (livello di trasporto) **fra due JVM** si cerca di **riutilizzarla**
- Diverse possibilità
 1. Apro una sola connessione e la utilizzo per servire una richiesta alla volta → forti effetti di **sequenzializzazione delle richieste**
 2. Utilizzo la connessione aperta se non ci sono altre invocazioni remote che la stanno utilizzando; altrimenti ne apro una nuova → **maggior impiego di risorse** (connessioni), ma **effetti di sequenzializzazione mitigati**
 3. Utilizzo **una sola connessione** (al livello di trasporto) per servire diverse richieste, e su quella faccio del **demultiplexing** per l'invio delle richieste e la ricezione delle risposte

↑
Sequenzializzazione

Distribuzione delle classi (deployment)

- In una applicazione RMI è necessario che siano disponibili gli opportuni file **.class** nelle località che lo richiedono (per l'esecuzione o per la de/serializzazione)
- Il **Server** deve poter accedere a:
 - interfacce che definiscono il servizio → a tempo di compilazione
 - implementazione del servizio → a tempo di compilazione
 - stub e skeleton delle classi di implementazione → a tempo di esecuzione
 - altre classi utilizzate dal server → a tempo di compilazione o esecuzione
- Il **Client** deve poter accedere a:
 - interfacce che definiscono il servizio → a tempo di compilazione
 - stub delle classi di implementazione del servizio → a tempo di esecuzione
 - classi del server usate dal client (es. valori di ritorno) → a tempo di compilazione o esecuzione
 - altre classi utilizzate dal client → a tempo di compilazione o esecuzione

Classpath ed esecuzione

Rmiregistry, server e client devono poter accedere alle classi necessarie per l'esecuzione. Si presti quindi particolare attenzione al **direttorio** dove vengono lanciati il registry, il server e il client

In particolare, ipotizzando di avere tutti i file .class nel direttorio corrente (“.”), e di lanciare registry, client, e server dal direttorio corrente, bisogna **aggiungere al CLASSPATH tale direttorio**

Sotto Linux: ciò è possibile **aggiungendo nella propria directory HOME il file ".profile" (creandolo se non esiste)**. In particolare, il file .profile deve contenere le seguenti linee per aggiungere il direttorio corrente al CLASSPATH:

```
CLASSPATH=.:$CLASSPATH
```

```
export CLASSPATH
```

Si noti che questa è la modalità standard in Linux per aggiungere/modificare una variabile di ambiente.

Nelle FAQ del corso, si veda anche **il caso della variabile di ambiente PATH**

E se volessimo lanciare il client, il server, e il registry in direttori diversi?

RMI Class loading

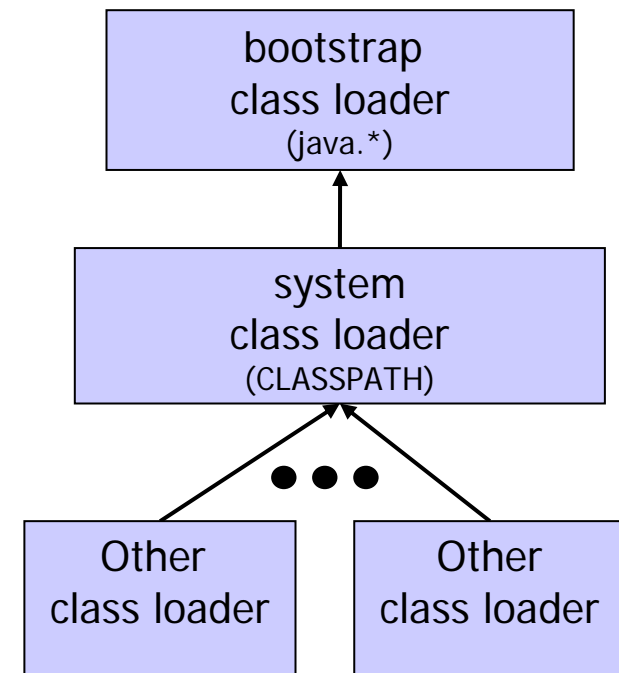
In Java si definisce un ***ClassLoader***, cioè una **entità capace di risolvere i problemi di caricamento delle classi** dinamicamente e di riferire le classi ogni volta che ce ne sia necessità, oltre che di rilocarle in memoria

Le Classi possono sia essere caricate dal disco locale e dalla rete (vedi applet) con vari **gradi di protezione**

Java consente di definire una **gerarchia di *ClassLoader*** diversi, ciascuno responsabile del caricamento di classi diverse, e anche definibili dall'utente

I ClassLoader sono una località diversa l'uno dall'altro e non comunicano uno con l'altro: possono avere anche visioni non consistenti

Enforcing fatto da Security Manager



Sicurezza in RMI

- Sia il client che il server devono essere lanciati specificando il **file con le autorizzazioni** (file di policy) consultato dal security manager (**entità per il controllo dinamico della sicurezza**)
- Per l'esecuzione sicura del codice si richiede l'utilizzo del **RMI SecurityManager**
 - RMI SecurityManager effettua il **controllo degli accessi** (specificati nel **file di policy**) alle risorse di sistema e **blocca gli accessi non autorizzati**
 - Il security manager viene creato **all'interno dell'applicazione RMI** (sia **lato client**, sia **lato server**), se non ce n'è già uno istanziato

```
if (System.getSecurityManager() == null)
{System.setSecurityManager(new RMI SecurityManager()); }
```
- Esempio
 - Client: `java -Djava.security.policy=echo.policy EchoRMIClient`
 - Server: `java -Djava.security.policy=echo.policy EchoRMIServer`

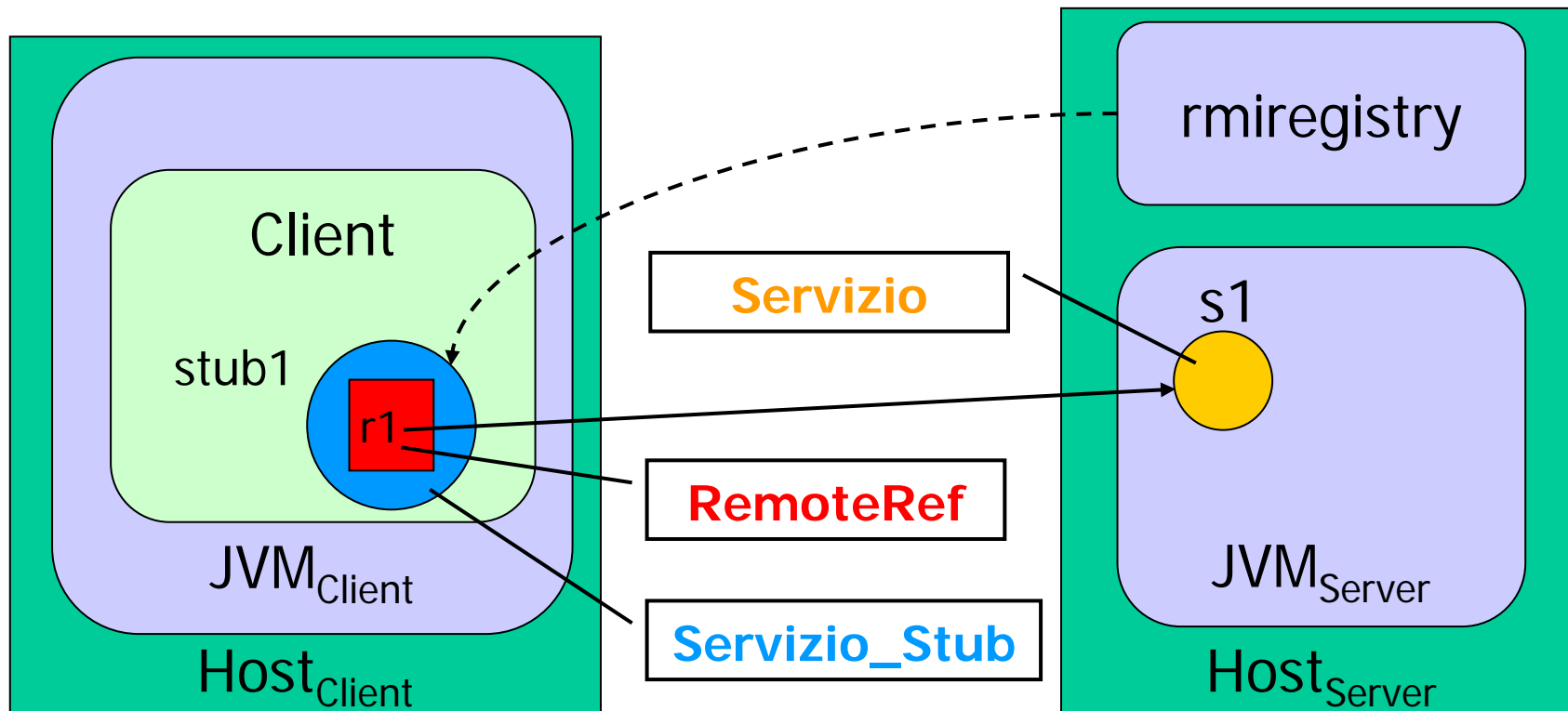
File di policy

- Struttura del file di policy:

```
grant {  
    permission java.net.SocketPermission "*:1024-65535", "connect, accept";  
    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 permesso consente di **prelevare il codice** da un **server http**
- Il terzo permesso consente di **prelevare codice** a partire dalla **radice dei direttori consentiti**

Stub e Riferimenti Remoti



Il **Client** accede al **Server RMI** implementato dalla classe **Servizio** attraverso il riferimento allo stub locale *stub1* (istanza della classe **Servizio_Stub** e passata dal registry al client)

Servizio_Stub contiene al suo interno un **RemoteRef** (*r1*) che consente al RRL di raggiungere il server

RMI Registry: il problema del bootstrap

Come avviene l'avvio del sistema (**bootstrap**) e ritrovare il riferimento remoto?

- Java mette a disposizione la classe **Naming**, che realizza dei metodi statici per effettuare le operazioni di de/registrazione e reperimento del riferimento del server
- I metodi per agire sul registry hanno bisogno dello stub del registry
- **come ottenere un'istanza dello stub del registry senza consultare un registry?**

Costruzione locale dell'istanza dello stub a partire da:

- **Indirizzo** server e **porta** contenuti nel nome dell'oggetto remoto
- Identificatore (locale alla macchina server) dell'oggetto registry fissato a priori dalla specifica RMI della SUN → **costante fissa**

Sicurezza e 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 da nodi esterni

N.B.: da ciò segue che sull'host in cui vengono effettuate le chiamate al registry deve essercene sempre **almeno uno in esecuzione**

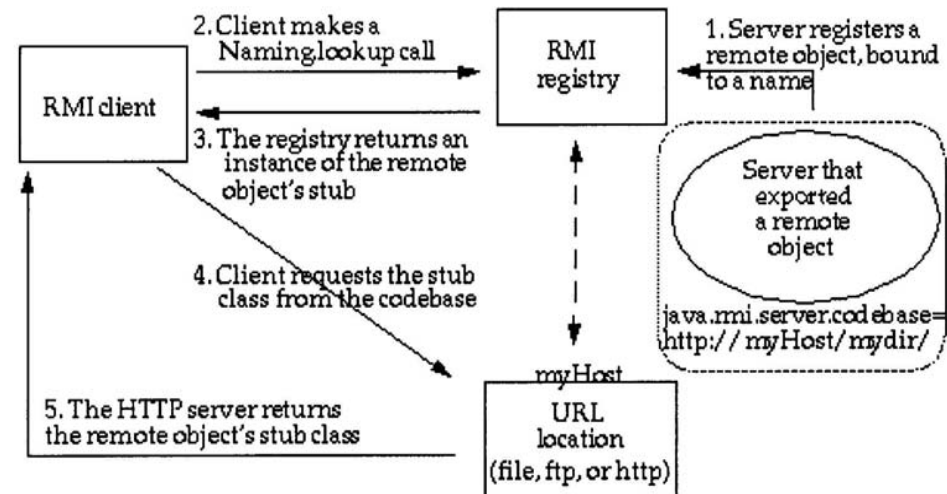
Localizzazione del codice

- È necessario:
 1. Localizzare il codice (in locale o in remoto)
 2. Effettuare il download (se in remoto)
 3. Eseguire in modo sicuro il codice scaricato
- Le informazioni relative a dove reperire il codice sono memorizzate sul server e **passate al client by need**
 - **Server** RMI mandato in esecuzione specificando nell'opzione **java.rmi.server.codebase** con l'URL da cui prelevare le classi necessarie
 - L'URL puo' essere
 - l'indirizzo di un server http (**http://**)
 - l'indirizzo di un server ftp (**ftp://**)
 - una directory del file system locale (**file://**)
- Il codebase è una proprietà del server che viene **annotata nel RemoteRef** pubblicato sul registry (cioè contenuta **nell'istanza dello stub**)
- Le classi vengono cercate sempre **prima nel CLASSPATH locale**, solo in caso di insuccesso vengono cercate nel codebase

Utilizzo del codebase

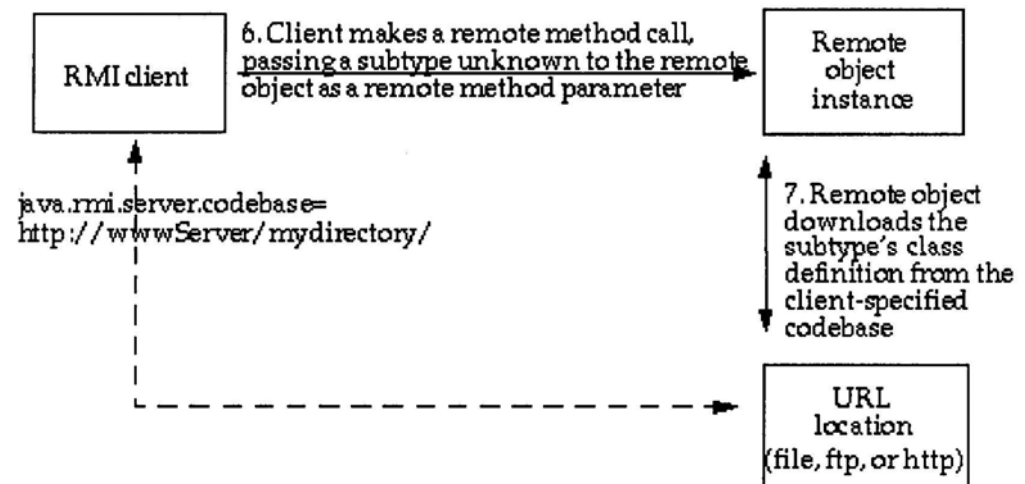
- Il codebase (contenuto nel RemoteRef) viene usato dal **client** per scaricare le classi necessarie relative al server (interfaccia, stub, oggetti restituiti come valori di ritorno)

- **NOTA:** differenza fra **istanza** e **classe** dello stub

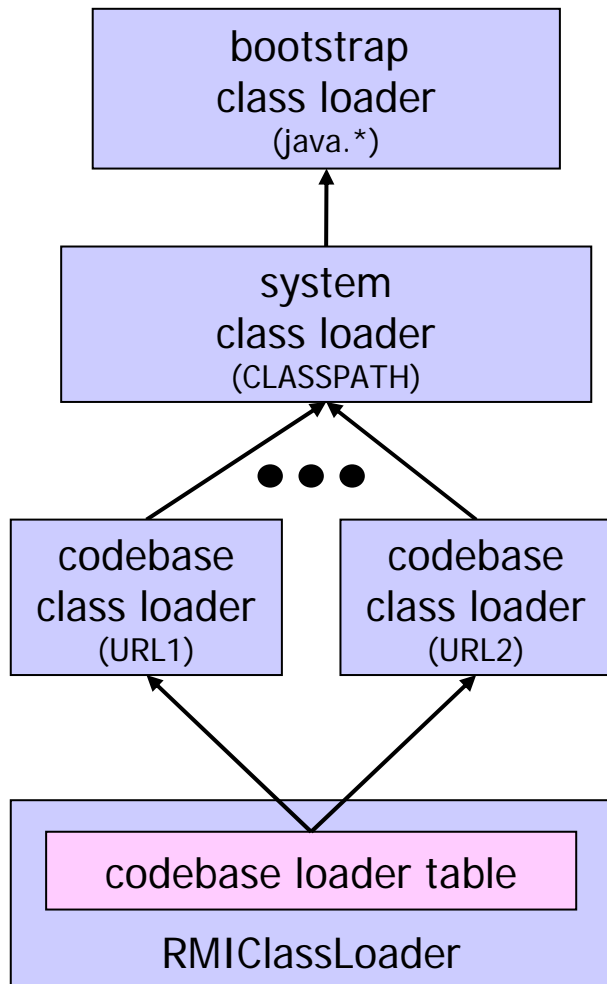


- Cosa accade per il passaggio per valore (dal client al server) di un oggetto che sia istanza di una classe non nota al server?

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



RMI Class loading



- **ClassLoader**: risolve i nomi delle classi nelle definizioni delle classi (codice – bytecode)
- Java definisce una **gerarchia di ClassLoader** diversi, ciascuno responsabile del caricamento di classi diverse, e anche definibili dall'utente
- **Codebase classloader** di Java RMI: responsabili del caricamento di classi raggiungibili con un qualsiasi URL standard (codebase) → **anche da remoto**
- **RMIClassLoader NON** un ClassLoader vero e proprio, ma un componente di supporto RMI che esegue 2 operazioni fondamentali:
 - Estrae il campo **codebase** dal riferimento dell'oggetto remoto
 - Usa i **codebase classloader** per caricare le classi necessarie dalla locazione remota

Bibliografia

- Sito della Sun:
 - <http://java.sun.com/products/jdk/rmi/>
- W.Grosso, “**Java RMI**”, Ed. O'Reilly, 2002
- R. Öberg, “**Mastering RMI, Developing Enterprise Applications in Java and EJB**”, Ed. Wiley, 2001
- M. Pianciamore, “**Programmazione Object Oriented in Java: Java Remote Method Invocation**”, 2000

Per contattare Luca Foschini:

- E-mail: luca.foschini@unibo.it
- Home page: www.lia.deis.unibo.it/Staff/LucaFoschini