



Università degli Studi di Bologna
Scuola di Ingegneria

Corso di Reti di Calcolatori T

Java RMI
(Remote Method Invocation)

Antonio Corradi

Anno accademico 2023/2024

RMI: PRODROMO

Cosa è una Interfaccia a confronto con una Classe?

Sono entrambi descrittori di istanze (entità di metalivello)

Uno concreto (con metodi), l'altro astratto

Ereditarietà tra enti descrittivi

Relazione di derivazione di comportamenti da altre entità descrittive dello stesso tipo: chi eredita ha tutto quello specificato dalla super entità

Tra classi ereditarietà

singolo genitore

Tra interfacce

ereditarietà multipla

RMI: PRODROMO

Interfaccia e Classe sono entrambe entità di metalivello,
ossia descrittive delle istanze

Ogni istanza riferisce una ed una sola classe (da cui è creata)

Una classe implementa molte interfacce

Ogni istanza fa riferimento a molte interfacce (dalla sua classe)

Tipi di variabili (a parte variabili di classe) nelle istanze

Variabili nelle istanze (primitivi o riferimenti ad istanze)

Variabili interne ad un oggetto istanza sono tipizzate
(con tipi primitivi o classi)

Variabili interfaccia (tipizzate dalla interfaccia)

Variabili interfaccia possono puntare ad una istanza di una classe che implementi la interfaccia stessa

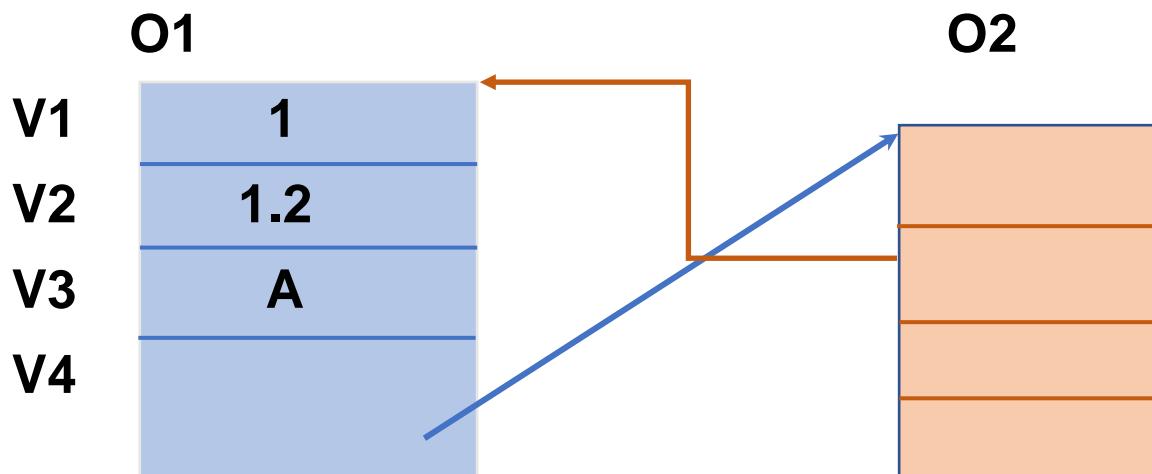
RMI: PRODROMO

Che entità / oggetti ci sono in Java? A run-time
Classi? Istanze? Interfacce? tutti

Come sono le istanze? Semantica per riferimento

Una istanza contiene le variabili descritte dalla classe e riferisce i metodi nella classe (valori per primitivi e riferimenti per altri oggetti)

Gli oggetti non sono contenuti dentro gli oggetti ma si puntano tra loro creando un grafo per ogni oggetto (semantica locale per riferimento)



RMI: MOTIVAZIONI E GENERALITÀ

La architettura RMI introduce la possibilità di **richiedere esecuzione di metodi remoti in JAVA (RPC in JAVA)** integrando il tutto con il **paradigma OO (object-oriented)**

Definizioni e generalità (RMI e diverse JVM)

RMI come 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 localmente mantenuto in una variabile interfaccia**

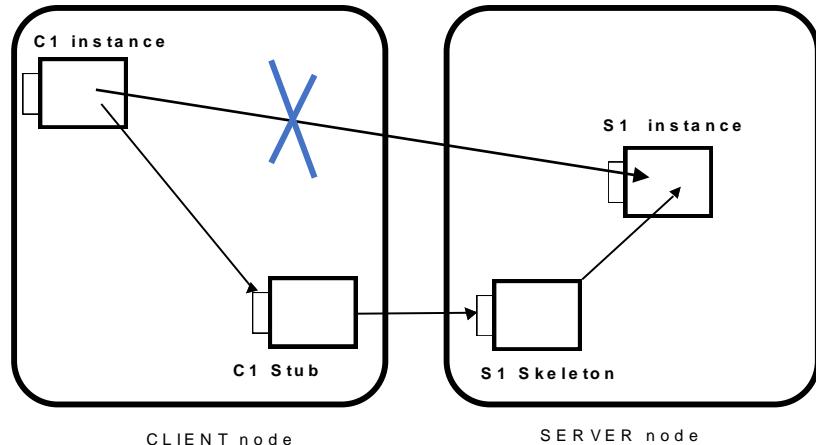
Unico ambiente di lavoro come conseguenza del linguaggio Java, ma
Eterogeneità di sistemi → grazie alla **portabilità del codice Java (BYTECODE)**

ACCESSO AD OGGETTI REMOTI

In Java **non sono (direttamente) disponibili riferimenti remoti**, ma **RMI** permette di **costruirli**

- Remote Method Invocation
 - Due **proxy: stub** dalla parte cliente e **skeleton** dalla parte servitore
 - **Pattern Proxy**: questi componenti **nascondono** al livello applicativo **la natura distribuita dell'applicazione**

Cambiano **le cose 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

TRUCCHI DI SUPPORTO

- In Java **sono compilate interfacce e classi?**
- In Java **sono presenti le classi a run-time?**
- In Java **sono presenti le interfacce a run-time?**

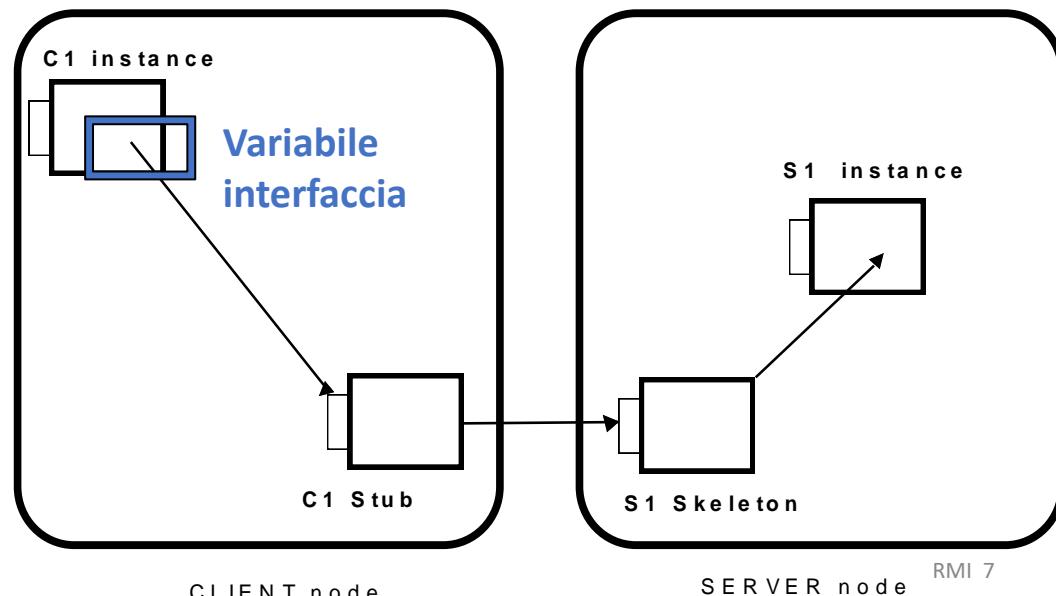


RMI ricorre alle variabili interfaccia

- Una **variabile interfaccia** è una variabile che dinamicamente può contenere (un riferimento a) **una istanza di una classe** (qualsiasi) che **implementa la interfaccia stessa**

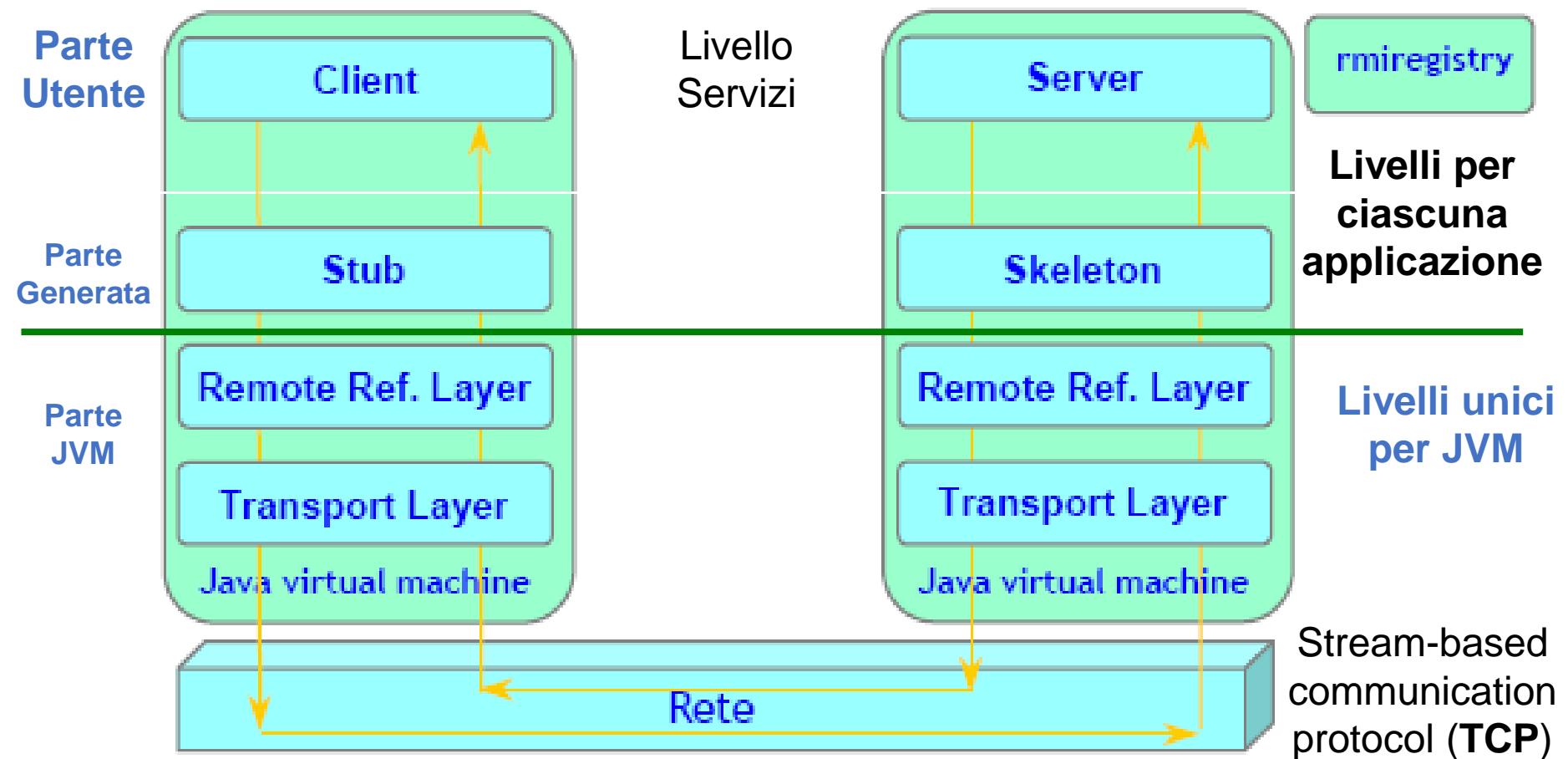
TRUCCO

Si usa una **variabile interfaccia** per contenere un **riferimento** a un proxy che permette di **controllare e preparare** il passaggio da un ambiente **cliente** ad un ambiente **servitore**



ARCHITETTURA RMI

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



ARCHITETTURA RMI A LIVELLI E COMPONENTI

Stub e skeleton (generati sotto il controllo dell'utente – vedi poi):

- **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

Il livello Remote Reference Layer (RRL):

- Responsabile della gestione dei **riferimenti agli oggetti remoti**, dei **parametri** e delle **astrazioni di una connessione stream-oriented**

I proxy sono entità intermedie ma presenti e da considerare

ARCHITETTURA RMI A LIVELLI E COMPONENTI

Il livello di **Trasporto connesso**, Transport Layer (TL)

- Responsabile della gestione delle connessioni fra i diversi spazi di indirizzamento (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**
- Il **sistema di nomi, Registry**: **servizio di nomi** che consente al server di pubblicare un servizio e al client di recuperarne il proxy

I due livelli, RRL e Transport, sono parte della macchina virtuale JVM

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 differente host
- Un **oggetto descritto tramite (una o più) interfacce remote** che dichiarano i metodi accessibili da remoto

Chiamata locale vs. chiamata remota

Il cliente invoca un **metodo** di un **oggetto remoto** attraverso un **riferimento remoto (variabile interfaccia)**

- **Sintassi: simile (circa uguale) → trasparenza** per il cliente
Chiamata **sincrona e bloccante** sempre con attesa
- **Semantica: diversa**
Chiamate locali → **affidabilità unica e atomica (tutto o niente)**
Chiamate remote → **affidabilità diversa (possibilità di fallimento)**

→ **semantica “at-most-once”** con uso TCP

Server remoto come locale: ogni chiamata esegue in modo **indipendente e parallelo (vedere poi i dettagli)**

INTERFACCE E IMPLEMENTAZIONE

Alcune considerazioni pratiche:

- **Separazione** tra
 - Definizione del comportamento → **interfacce**
 - Implementazione del comportamento → **classe**
- I **componenti remoti** sono riferiti tramite **variabili interfaccia** (che possono contenere solo istanze di classi che implementano la interfaccia)
- Alcuni vincoli sui **componenti**
 1. **Definizione** del comportamento, **interfaccia**
 - **interfaccia** deve estendere `java.rmi.Remote`
 - **ogni metodo** deve propagare `java.rmi.RemoteException`
 2. **Implementazione** del comportamento, **classe server** che
 - deve **implementare** l'interfaccia definita
 - deve **estendere** `java.rmi.UnicastRemoteObject`

PASSI PER L'UTILIZZO DI JAVA RMI

È necessario fare una serie di passi:

1. Definire **interfaccia** e **implementazione** del **componente server** utilizzabile in **remoto** (quindi **interfaccia** e **server**, non il cliente)
2. Compilare la **interfaccia** e la **classe** dette sopra (con **javac**), poi generare **stub** e **skeleton** (con **rmic**) della classe utilizzabile in remoto
3. Pubblicare il servizio nel **sistema di nomi**, **registry** o **RMIregistry**
 - attivare il **registry**
 - registrare il **servizio** (il server deve fare una bind sul registry)
4. Ottenere (**lato cliente**) il riferimento all'oggetto remoto tramite il **name service**, facendo una **lookup sul registry**, e **compilare il cliente**

A questo punto **l'interazione tra il cliente e il server** può procedere (prima si deve attivare il servitore poi il cliente)

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

IMPLEMENTAZIONE: INTERFACCIA

- Si deve definire la interfaccia remotizzabile
- L'interfaccia deve **estendere l'interfaccia Remote**
- Ciascun metodo remoto
 - Deve lanciare una **RemoteException** in caso di problemi; quindi l'invocazione dei metodi remoti **NON è** completamente trasparente
 - Ha **un solo** risultato di **uscita e nessuno, uno o più** parametri di **ingresso (1 parametro in in e non out)**
 - I parametri **devono** essere passati per **valore** (dati **primitivi** o oggetti **Serializable**) o per **riferimento remoto** (oggetti **Remote**) → questo 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: INTERFACCIA

EchoInterface

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

- La interfaccia deve essere remotizzabile ed estendere **Remote**
- Ogni metodo deve potere mandare la **eccezione Remote che segnala errore remoto o problemi di comunicazione**

IMPLEMENTAZIONE: SERVER

Si deve definire la implementazione del server

La **classe che implementa il server**

- Deve estendere la classe UnicastRemoteObject
- Deve implementare **tutti i metodi definiti nell'interfaccia**

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

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

Registrazione del servizio

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

```
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; }  
  
    public static void main(String[] args){  
        // Registrazione del servizio  
        try  
        {  
            EchoRMIServer serverRMI =  
                new EchoRMIServer();  
            Naming.rebind(  
                "//localhost:1099/EchoService", serverRMI);  
        }  
        catch (Exception e)  
        {e.printStackTrace(); System.exit(1); }  
    }  
}
```

IMPLEMENTAZIONE: ISTANZA SERVER

EchoServer main per la registrazione al RMIServer

```
public static void main(String args[])
{try
{EchoRMIServer serverRMI = new EchoRMIServer();
Naming.rebind("//localhost/1099/EchoService",
serverRMI);

}
catch (Exception e)
{System.out.println("Error: " + e.getMessage());
e.printStackTrace(); system.exit();
}
}
```

- Creazione istanza del **server serverRMI**
- Registrazione al sistema di nomi **RMIServer** locale

IMPLEMENTAZIONE: CLIENT

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

Reperimento di **un riferimento**
remoto (con una **lookup**) e
memorizzandolo in **una**
variabile di tipo interfaccia
Nodo **server anche non locale**

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(
                    "//localhost:1099/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); }
    }
}
```

IMPLEMENTAZIONE: CLIENTE

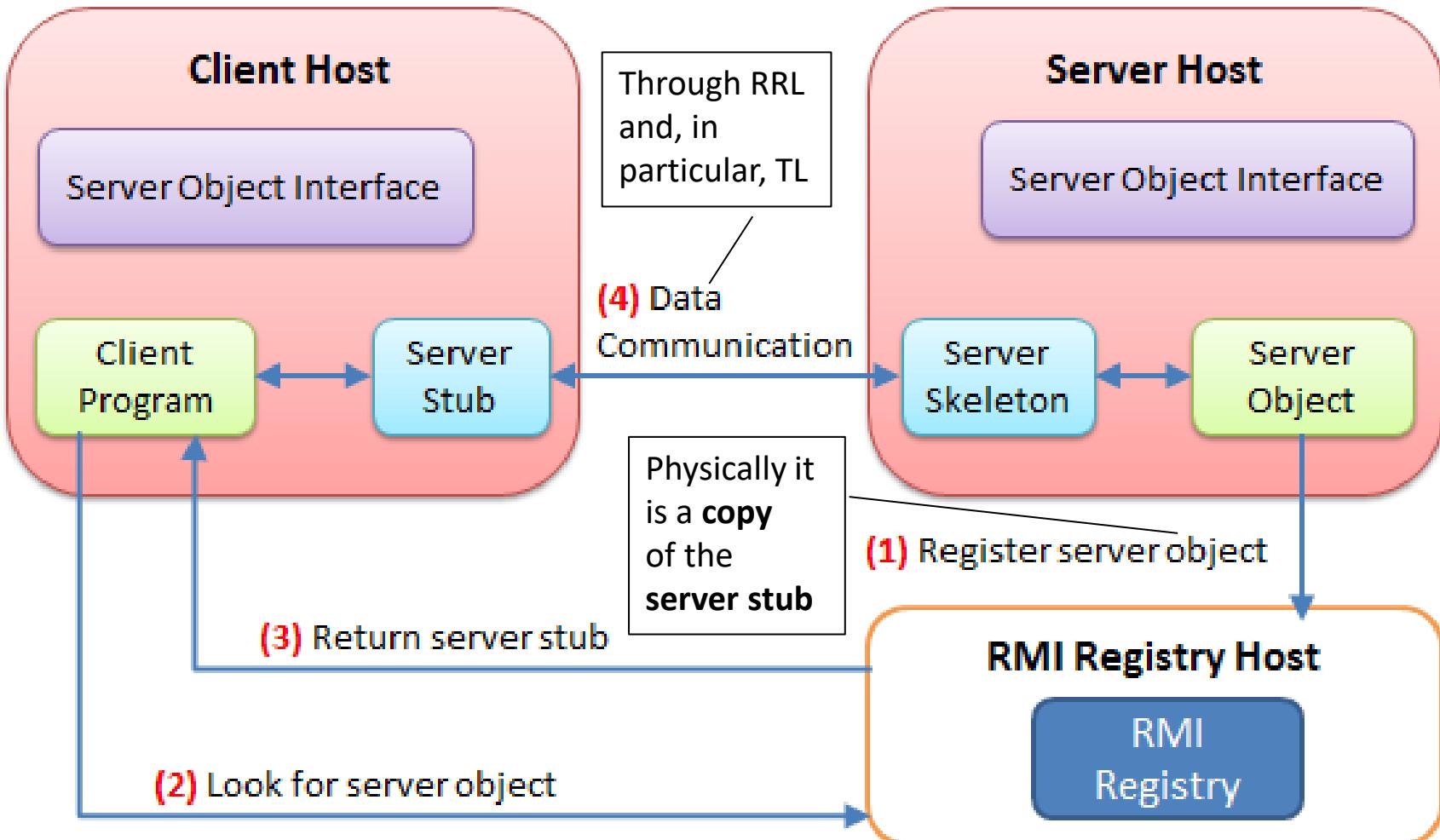
Il Cliente ottiene la variabile remota per fare le chiamate

```
public static void main(String args[])
{try {...  
EchoInterface serverRMI = (ServerRMI)
Naming.lookup("//remotehost/1099/EchoService",
serverRMI); ...  
  
... s = serverRMI.getEcho(mess) ...  
}  
  
catch (Exception e)
{System.out.println("Error: " + e.getMessage());
e.printStackTrace(); system.exit();
}}}
```

Creazione variabile **interfaccia remota ServerRMI**
andando al **nodo remoto**

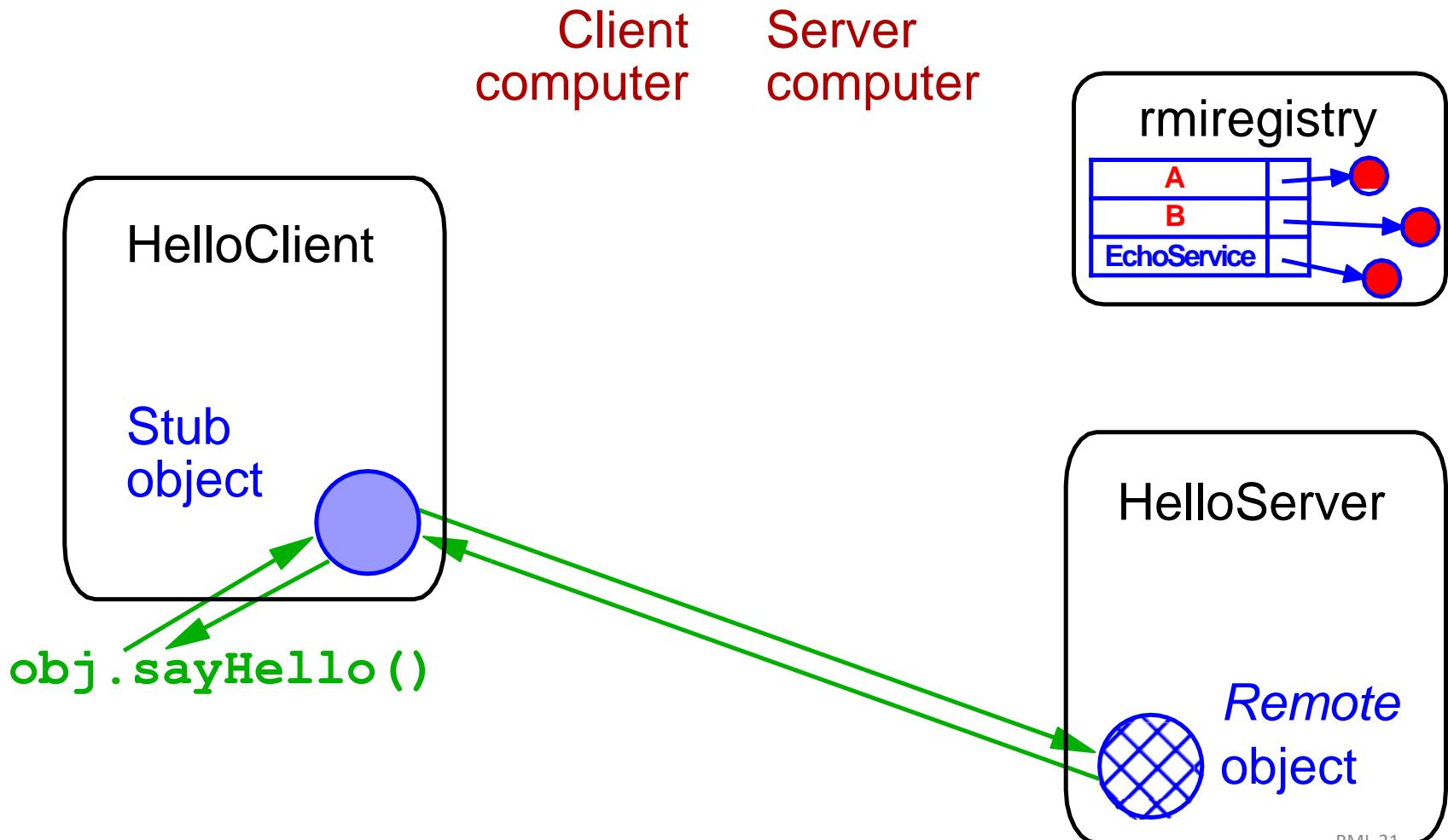
- Invocazioni **remote successive**

VISIONE DI INSIEME



VISIONE DI INSIEME

Esempio: Cliente e Server (RMiregistry) e chiamata

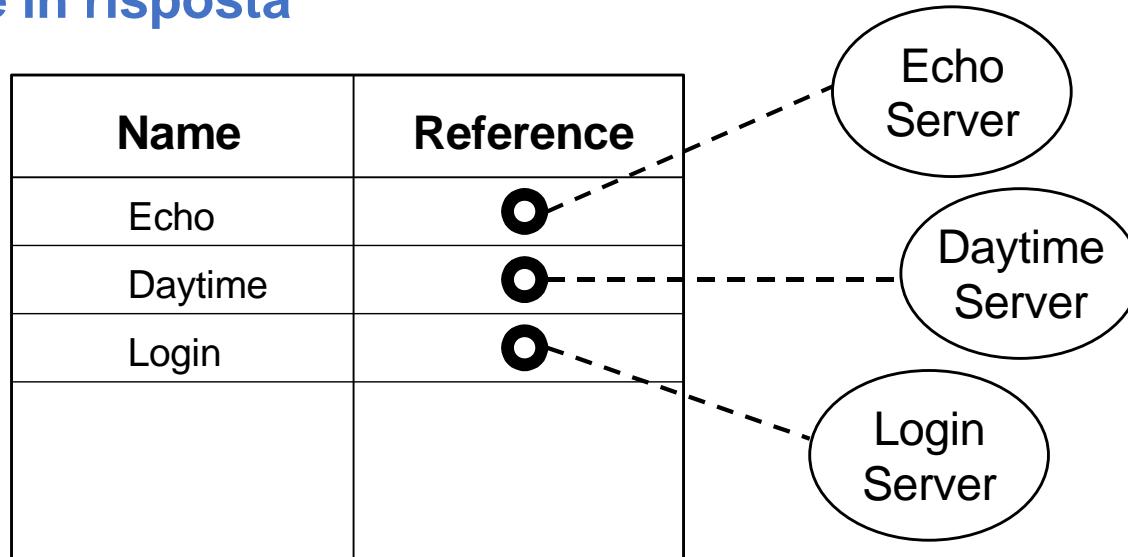


RMI REGISTRY

Localizzazione del servizio: un client in esecuzione su una macchina ha bisogno di **localizzare un server** a cui connettersi, che è in esecuzione **in una JVM su una macchina diversa nota al cliente**

Varie possibili soluzioni:

- Il client **conosce in anticipo** dov'è allocato il **server**
- L'utente dice all'applicazione client dov'è **allocato il server** (ad es. via e-mail al client o con un **argomento di invocazione**)
- Un servizio **standard (naming service)** in una locazione ben nota, che il client conosce, funziona come **punto di indirezione e fornisce la locazione in risposta**

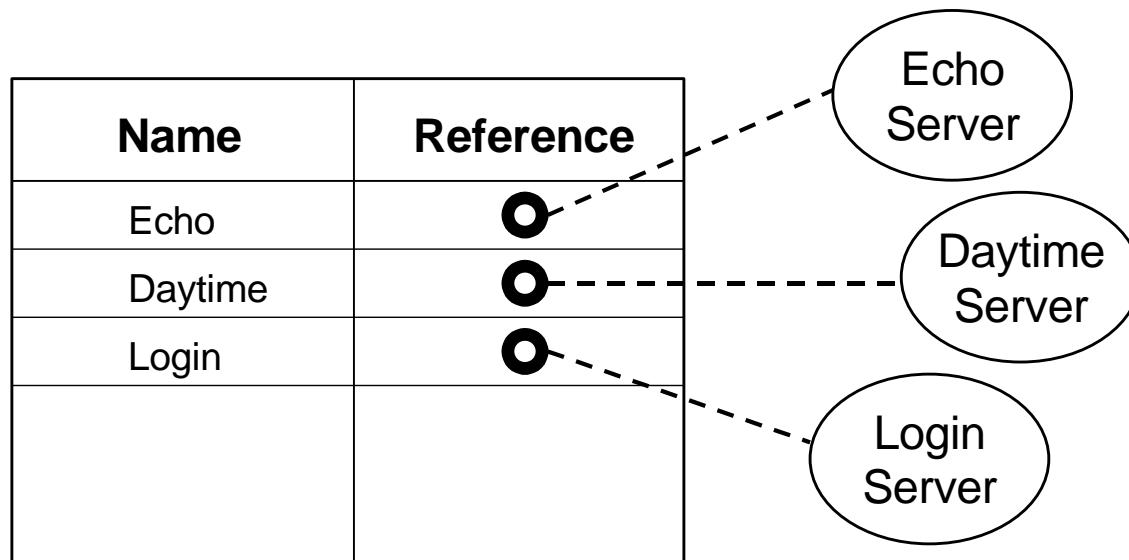


RMI REGISTRY

Java RMI utilizza un naming service detto **RMI Registry**
Il Registry mantiene un insieme di coppie **{name, reference}**

name: **stringa arbitraria** non interpretata da JVM tipo
//nomehost:porta/servizio

NON c'è trasparenza alla locazione in questo nome interno
(nodo locale che diventerà remoto per i clienti)



CLASSE NAMING E ATTIVAZIONE REGISTRY

Attivazione registry (sull'host **server**): usare il programma **rmiregistry** di Sun lanciato in una shell a parte specificando o meno la porta (default 1099), invocando: **rmiregistry** oppure **rmiregistry 10345**

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

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()
- public **static** Remote lookup(String **name**)

Ognuno di questi metodi **richiede la informazione al RMI registry** identificato con **host** e **porta** come locazione

name ⇒ combina la **locazione** del registry e il **nome logico** del servizio, nel formato: **//registryHost:port/service_name**

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

Senza trasparenza alla locazione!!

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**
per la produzione dei proxy

3. Esecuzione lato server (registry e server)

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

Lato client

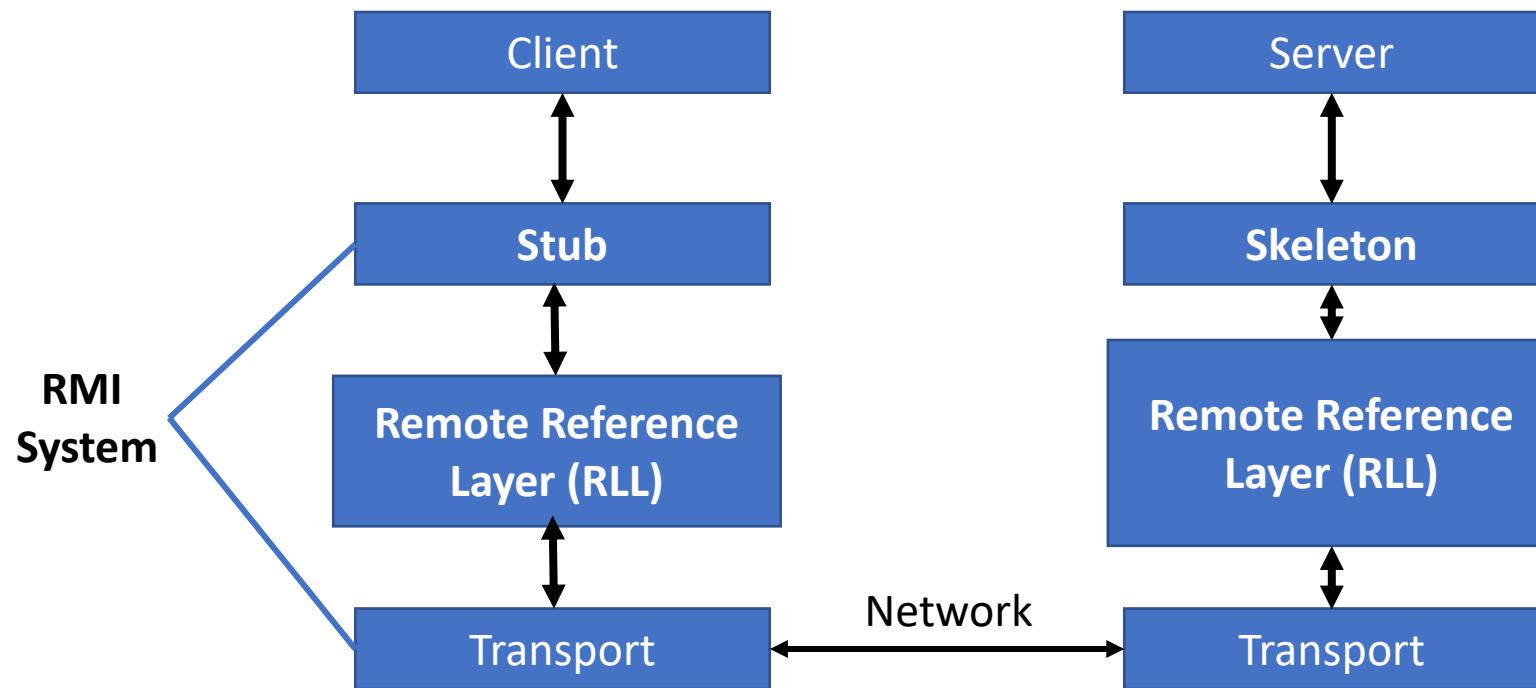
1. Compilazione: **javac EchoRMIClient.java**
2. Esecuzione: **java EchoRMIClient**

STUB E SKELETON

Rendono possibile la chiamata di un servizio remoto come se fosse locale (agiscono da proxy).

Sono generati dal **compilatore RMI**

L'ambiente **Java** supporta **direttamente** già la **de/serializzazione** e la **interfaccia Serializable**



STUB E SKELETON

Procedura di comunicazione in atto

1. il client ottiene **un riferimento remoto** e consente la interazione **C/S**
2. **attua la chiamata** dei **metodi** via stub e **aspetta il risultato**

Lo **stub**:

- effettua la **serializzazione** delle informazioni per la chiamata (id del metodo – identificazione - e argomenti)
- invia le informazioni allo skeleton utilizzando le astrazioni messe a disposizione dal **RRL**

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 allo stub

lo **stub**:

- effettua la **de-serializzazione** del valore di ritorno
- restituisce il **risultato** al client

DISTRIBUZIONE DELLE CLASSI (DEPLOYMENT) - SERVER

In RMI come ovvio, le fasi sono sia di sviluppo sia di esecuzione...

- In una applicazione RMI è necessario che siano disponibili a run-time 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 **eventuali classi** utilizzate dal server → a tempo di compilazione o esecuzione

DISTRIBUZIONE DELLE CLASSI (DEPLOYMENT) - CLIENT

In RMI come ovvio, le fasi sono sia di sviluppo sia di esecuzione...

- In una applicazione RMI è necessario che siano disponibili a run-time gli opportuni file `.class` nelle località che lo richiedono (per l'esecuzione o per la de/serializzazione)
- 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



Attenzione al directory dove vengono lanciati il registry, il server e il client

Ipotizzando di avere tutti i file .class nel directory corrente ("."), e di lanciare registry, client, e server dal directory corrente, **bisogna aggiungere alla variabile CLASSPATH il directory stesso**

Sotto Linux: aggiungere nella propria directory **HOME** il file di **profilo** (creandolo se non esiste)

Il file di configurazione dell'utente **.profile** (**.bashrc**, ...) **deve** contenere le seguenti linee per aggiungere il directory corrente al CLASSPATH:

- **CLASSPATH=.:\${CLASSPATH}**
- **export CLASSPATH**

Questa è la **modalità standard in Linux per aggiungere o modificare una variabile di ambiente** (vedi caso della variabile di ambiente **PATH** nelle FAQ del corso)

APPROFONDIMENTI SU RMI

LA SERIALIZZAZIONE

In generale, nei sistemi RPC, ossia in cui riferiamo funzionalità remote, i parametri di ingresso e uscita subiscono una duplice trasformazione per risolvere problemi di rappresentazioni eterogenee (presentazione di OSI)

- **Marshalling**: processo di codifica degli argomenti e dei risultati per la trasmissione
- **Unmarshalling**: processo inverso di decodifica di argomenti e risultati ricevuti

Ricordiamo che abbiamo la uniformità del supporto Java, assumendo un unico ambiente di riferimento fornito dalla JVM, che ha la stessa rappresentazione dei dati su ogni architettura

LA SERIALIZZAZIONE

In Java, grazie all'uso del BYTECODE uniforme e standard, **NON c'è bisogno di un/marshalling**: i dati vengono semplicemente **serializzati/deserializzati, ossia inseriti in un messaggio in sequenza o in modo lineare**, utilizzando le funzionalità offerte **direttamente a livello di linguaggio**

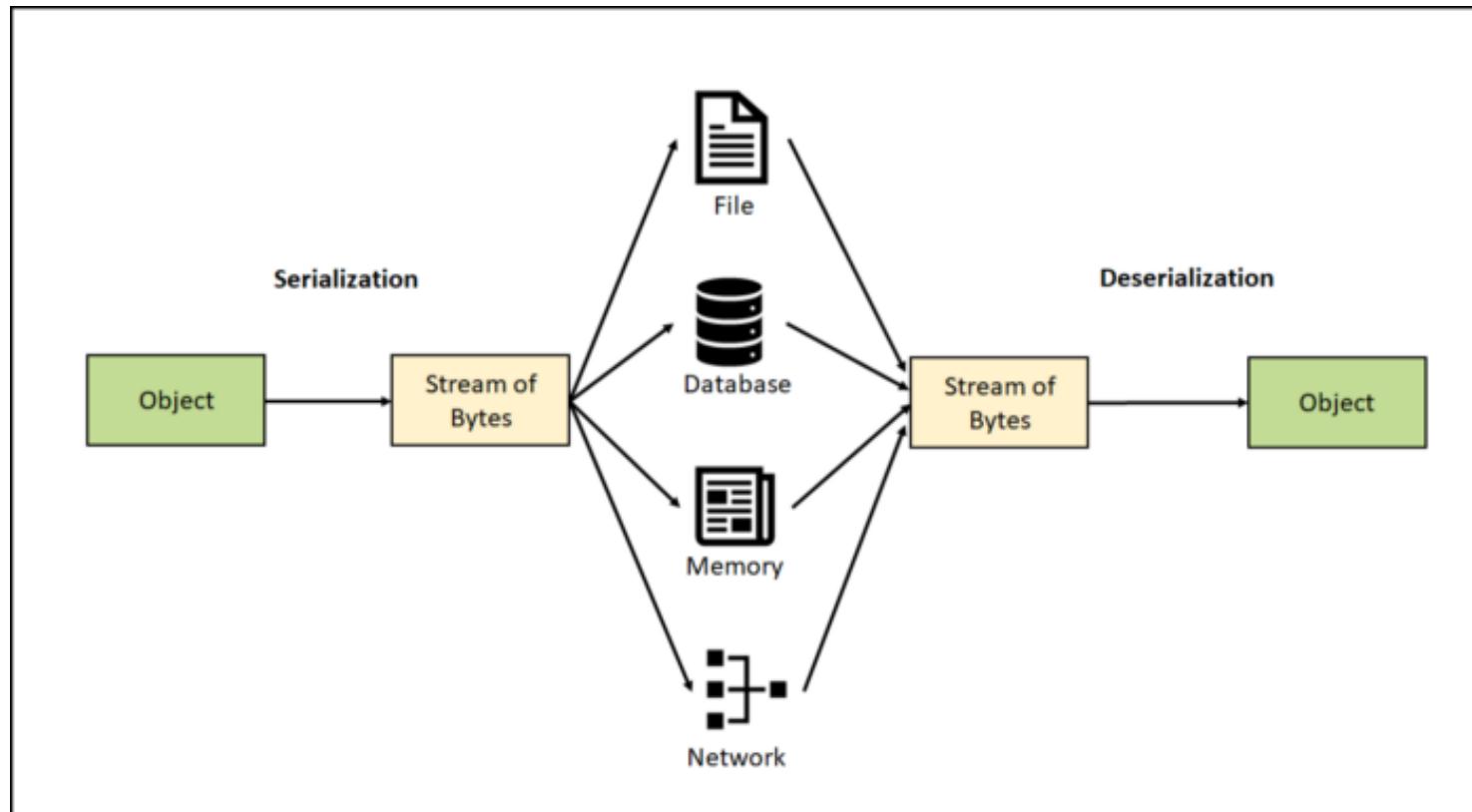
- **Serializzazione**: trasformazione di oggetti complessi in **semplici sequenze di byte** (anche di interi grafi di oggetti)
 - metodo `writeObject()` su uno stream di output
- **Deserializzazione**: decodifica di una sequenza di byte e costruzione di una copia dell'oggetto originale
 - metodo `readObject()` da uno stream di input

Stub e skeleton utilizzano queste funzionalità per lo scambio dei parametri di ingresso e uscita con l'host remoto

LA SERIALIZZAZIONE

In Java, grazie alle funzioni con stream di byte (per I/O) possiamo portare sul disco il contenuto unico di un oggetto e anche ripristinarlo da lì

- **Serializzando e**
- **Deserializzando**

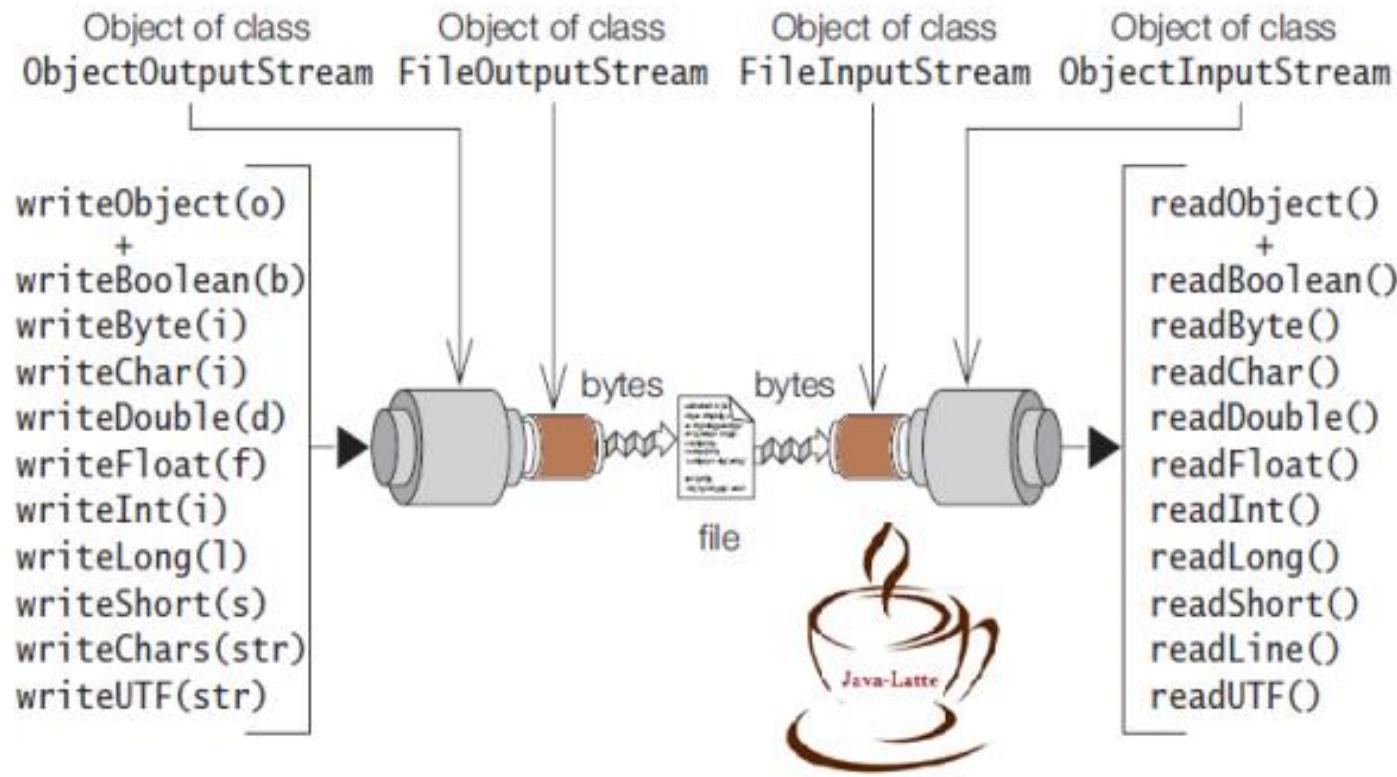


LA SERIALIZZAZIONE

In Java, grazie alle funzioni con stream di byte (per I/O) possiamo portare sul disco il contenuto unico di un oggetto e anche ripristinarlo da lì

- **Serializzando e**
- **Deserializzando**

Usando delle funzioni elementari di Output / Input



INTERAZIONE CON STREAM PER TX/RX

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

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

InputStream 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 (tutto il suo grafo) ma tutte le informazioni contenute che caratterizzano l’istanza singola e non la classe
no metodi, no costanti, no variabili static, no variabili transient

Al momento della deserializzazione sarà ricreata una copia dell’istanza “trasmessa” usando la classe locale ricevente (deve quindi essere accessibile!!!) dell’oggetto e tutte le informazioni ricevute per creare il grafo in remoto

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 ed è possibile ricostruirlo nell'ambiente locale**

Si consideri la possibilità che la classe locale non sia uguale a quella di partenza!!!!

Impaccamento e uso di un **hash per l'oggetto** che consente di verificare la correttezza **della classe sul nodo di arrivo** (le due classi devono avere lo stesso hash per procedere) per ogni istanza da trasferire

PASSAGGIO PARAMETRI IN LOCALE (NON RMI)

Tipo	Metodo Locale	Metodo Remoto
Tipi primitivi	Per valore	Per valore
Oggetti	Per riferimento	Per valore (interfaccia Serializable o deep copy , ossia copia dell'intero grafo a partire dalla istanza e a comprendere tutti gli oggetti riferiti)
Oggetti Remoti		Per riferimento remoto (interfaccia Remote), ossia viene passato lo stub all'oggetto remoto corrispondente

In **Locale** gli oggetti passati con semantica per riferimento:

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

PASSAGGIO DEI PARAMETRI – CASO RMI

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

In **Remoto** problema nel riferire entità e contenuti non locali

Passaggio per valore → tipi primitivi e Serializable Object

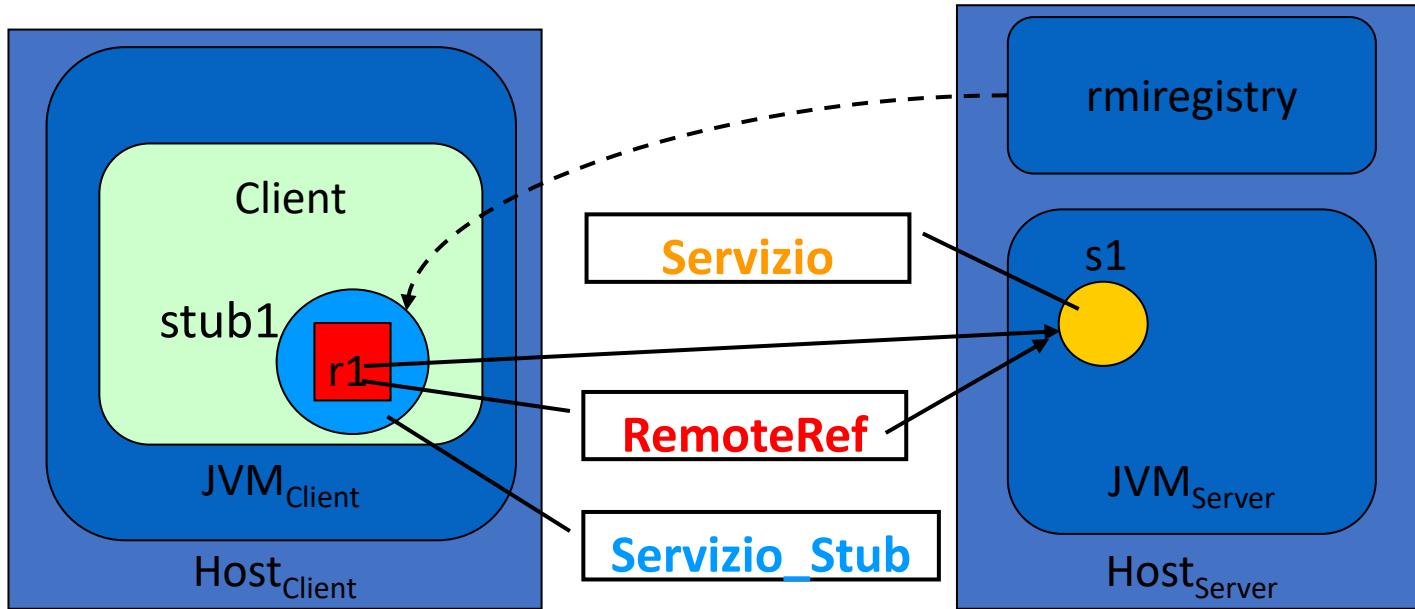
Oggetti la cui locazione **non è rilevante sono** passati **per copia ossia per valore**: ne viene serializzata l'istanza (grafo che parte dalla istanza) e trasmessa e deserializzata a destinazione per produrre una copia locale

Passaggio per riferimento remoto → Remote Object (via RMI)

Oggetti la cui funzione è **strettamente legata al nodo (e alla JVM) in cui eseguono** (server) sono passati **per riferimento remoto**: ne viene **serializzato lo stub**, creato automaticamente a partire dalla classe locale dello stub

STUB E RIFERIMENTI REMOTI

Ogni istanza di stub identifica l'oggetto remoto al quale si riferisce attraverso un **identificativo** (**ObjID**) che è **univoco** nella JVM dove l'oggetto remoto si trova e che viene mantenuto nello stub stesso



Il **Client** accede al **Server RMI** implementato dalla classe **Servizio** attraverso il riferimento remoto cioè lo *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

IMPLEMENTAZIONE DEL REGISTRY

Il **Registry** è in realtà **un server RMI** ☺ per ragioni di **economicità di implementazione e anche di principio**

- 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;  
}
```

IMPLEMENTAZIONE DEL REGISTRY

È anche possibile creare **all'interno del codice (del server)** un proprio **registry**:

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

che è un metodo della classe **LocateRegistry** (insieme ad altri)

In questo caso, il registry viene creato nella **stessa istanza** della **JVM del server**

STUB

Lo stub **effettua l'invocazione, gestisce la de/serializzazione, e spedisce/riceve gli argomenti e il risultato**

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**)
e un riferimento allo stato della chiamata (**remotecall**)

STUB

Intero che
indica il
metodo

```
...
// 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);
...
```

Hash della classe
del server

STUB

```
// 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

Lo **skeleton** **gestisce la de/serializzazione, spedisce/riceve i dati** appoggiandosi sul RRL, ed **invoca il metodo richiesto (dispatching)**

Metodo **dispatch** invocato dal RRL, con parametri d'ingresso

- Riferimento al server (**java.rmi.server.Remote**)
- Chiamata remota, numero operazione, e hash della classe server
- **NON c'è creazione di thread qui!**

Dove e da chi viene creato il thread????

SKELETON

```
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();  
            }  
    }  
}
```

SKELETON

```
// 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

Data la politica di generazione di thread, chi la specifica e la esprime?

Non a livello applicativo

(vedere **lo skeleton** → **non c'è generazione thread**)

Chi può esprimere la concorrenza e generare il thread?

E spegnerlo al termine del metodo



Tutto a livello di JVM (Livello Trasporto)

LIVELLO DI TRASPORTO: LA CONCORRENZA

Specifica aperta e non completa (EFFICIENZA su RISORSE)

Comunicazione e concorrenza sono aspetti chiave

Libertà di realizzare diverse implementazioni **ma:**

- **Implementazione** → **Server parallelo che deve essere thread-safe (uso di attività molteplici)**
- **Problematiche di sincronizzazione e di mutua esclusione** → **uso di lock: synchronized**

Uso di processi per ogni richiesta di servizio

RMI usa i **thread di Java** → **tipicamente generazione per call**

Questo implica di generare thread per ogni invocazione sull'oggetto remoto in esecuzione sulla JVM

Il livello Trasporto gestisce i 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**

Molte possibilità diverse, ma una sola implementata

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 **mitigando gli effetti di sequenzializzazione**
3. **Utilizzo una sola connessione** (al livello di trasporto) per servire diverse richieste, e su quella unica **demultiplexing** per l'invio delle richieste e la ricezione delle risposte

La connessione vive a livello di connessione JVM e permane fino alla chiusura delle JVM

Sequenzializzazione



CARICAMENTO CLASSI E APPLET

In Java **le classi possono essere caricate dinamicamente e da remoto** e solo al momento del bisogno, **durante la esecuzione**

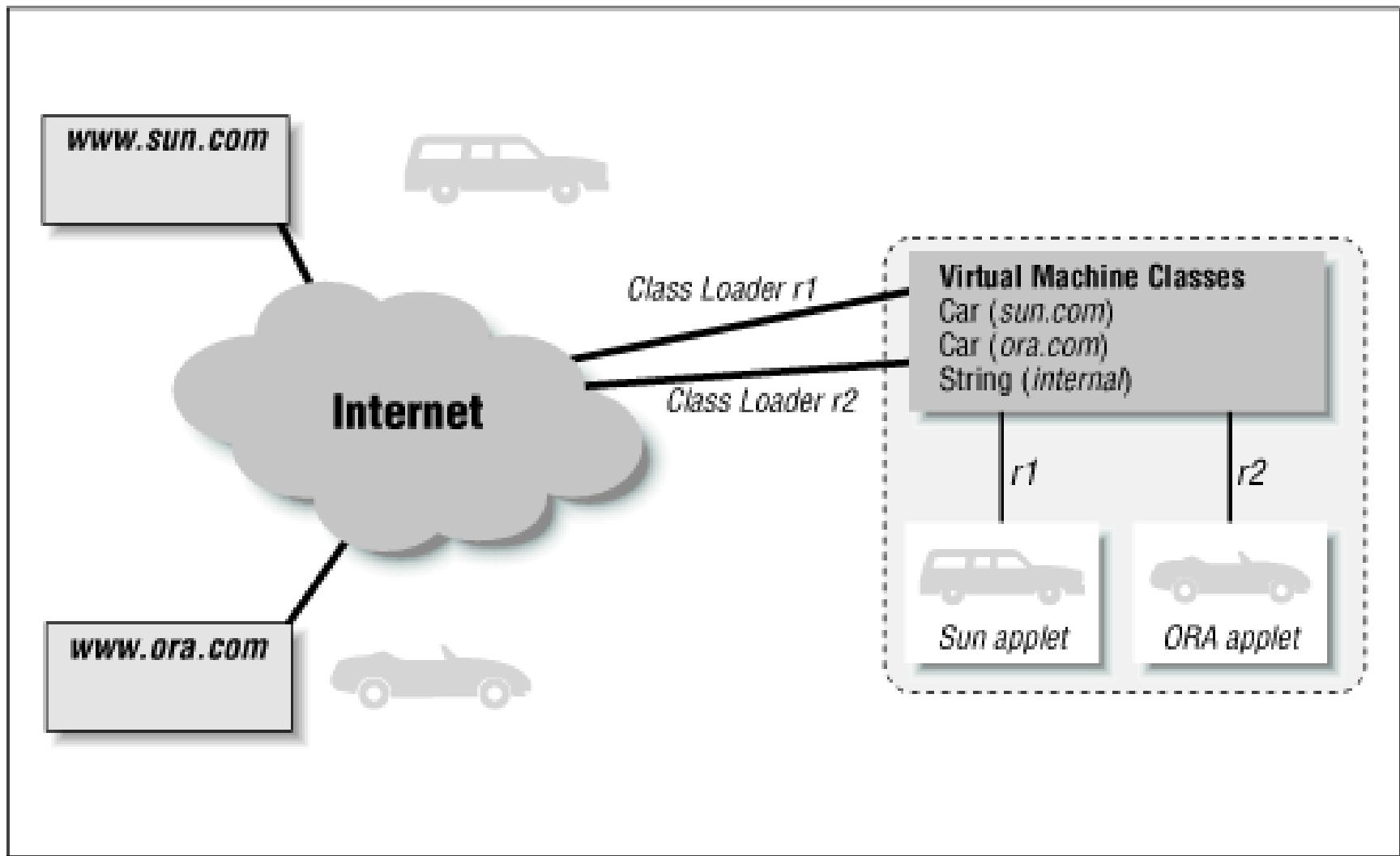
Java è fortemente integrato col Web e consente di associare alle pagine HTML anche **componenti Java compilati** che possono essere scaricati allo scaricamento della pagina Web

- Le **Applet** sono **componenti associati a pagine Web** che vengono portati al cliente e lì messi in esecuzione automaticamente: lo scaricamento è attuato per la presenza di una JVM con un **ClassLoader** associato al browser

Ci sono dei pericoli nello scaricamento di codice da un server Web non fidato?

- Quando scarichiamo dalla rete, il codice viene associato ad un **controllore delle operazioni** che può bloccare quelle non consentite (Security Manager) durante l'esecuzione prima di ogni operazione
- Le prime Applet avevano un **Security Manager con politica 'sandbox'** che impediva loro ogni **azione distruttiva** permettendo una esecuzione controllata e sicura

CARICAMENTI DINAMICI

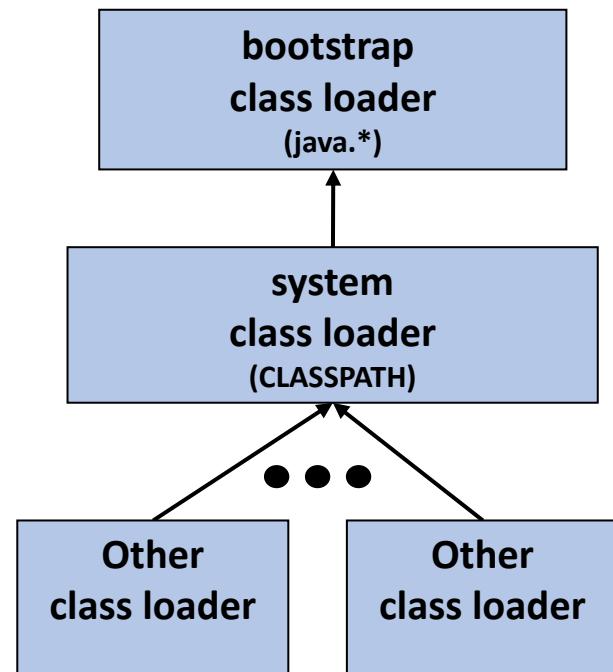


RMI CLASS LOADING

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

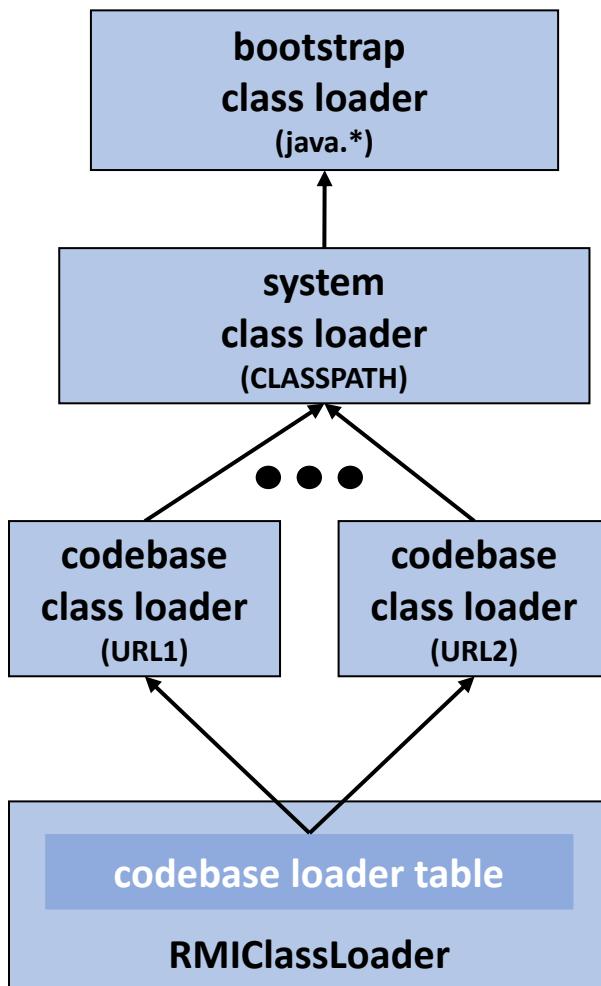
Le Classi possono sia essere caricate dal disco locale o dalla rete (vedi **applet**) con vari gradi di protezione

- Java consente di definire una gerarchia di **ClassLoader** diversi, ciascuno responsabile del caricamento di classi diverse, e anche definibili dall'utente
- I ClassLoader costituiscono ciascuno una località diversa l'uno dall'altro e non interferiscono tra loro: possono avere anche visioni non consistenti



Enforcing di sicurezza fatto da un Security Manager

RMI CLASS LOADING



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

- **Classloader**: risolve i nomi delle classi nelle definizioni delle classi (codice – bytecode)
- **Codebase classloader** di Java RMI: responsabili del caricamento di classi raggiungibili con un qualsiasi URL standard (codebase) → **anche da remoto**

RMIClassLoader **NON** è 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

LOCALIZZAZIONE DEL CODICE

Nel caso sia necessario ottenere dinamicamente del codice (stub o classi) **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 può essere l'indirizzo di un server http (`http://`) o ftp (`ftp://`) oppure 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**

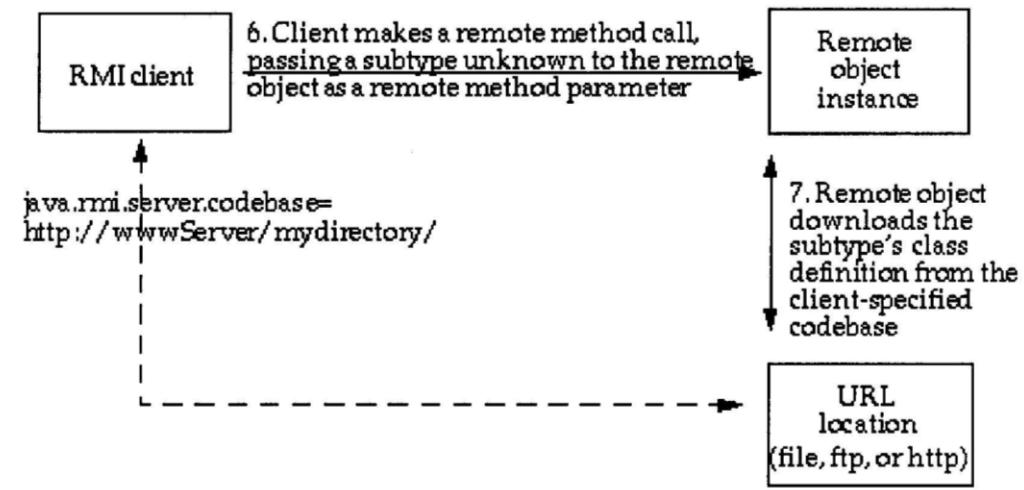
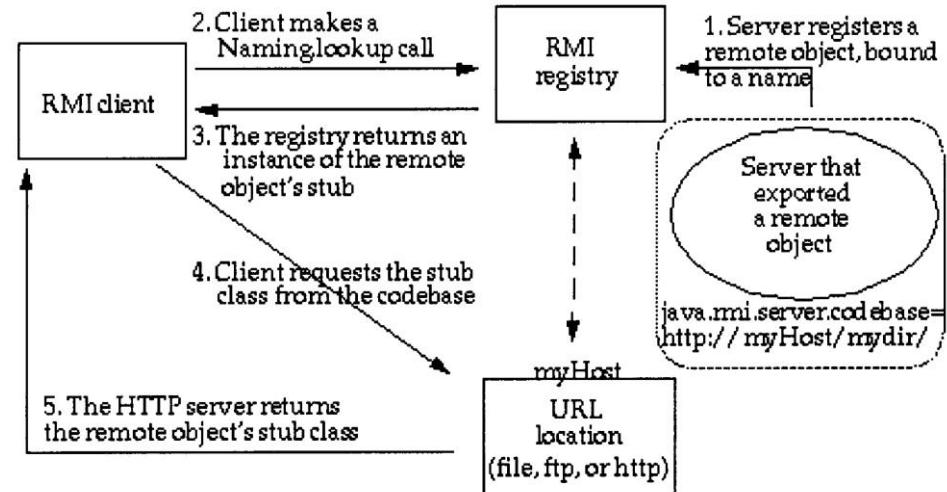
UTILIZZO DEL CODEBASE

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)

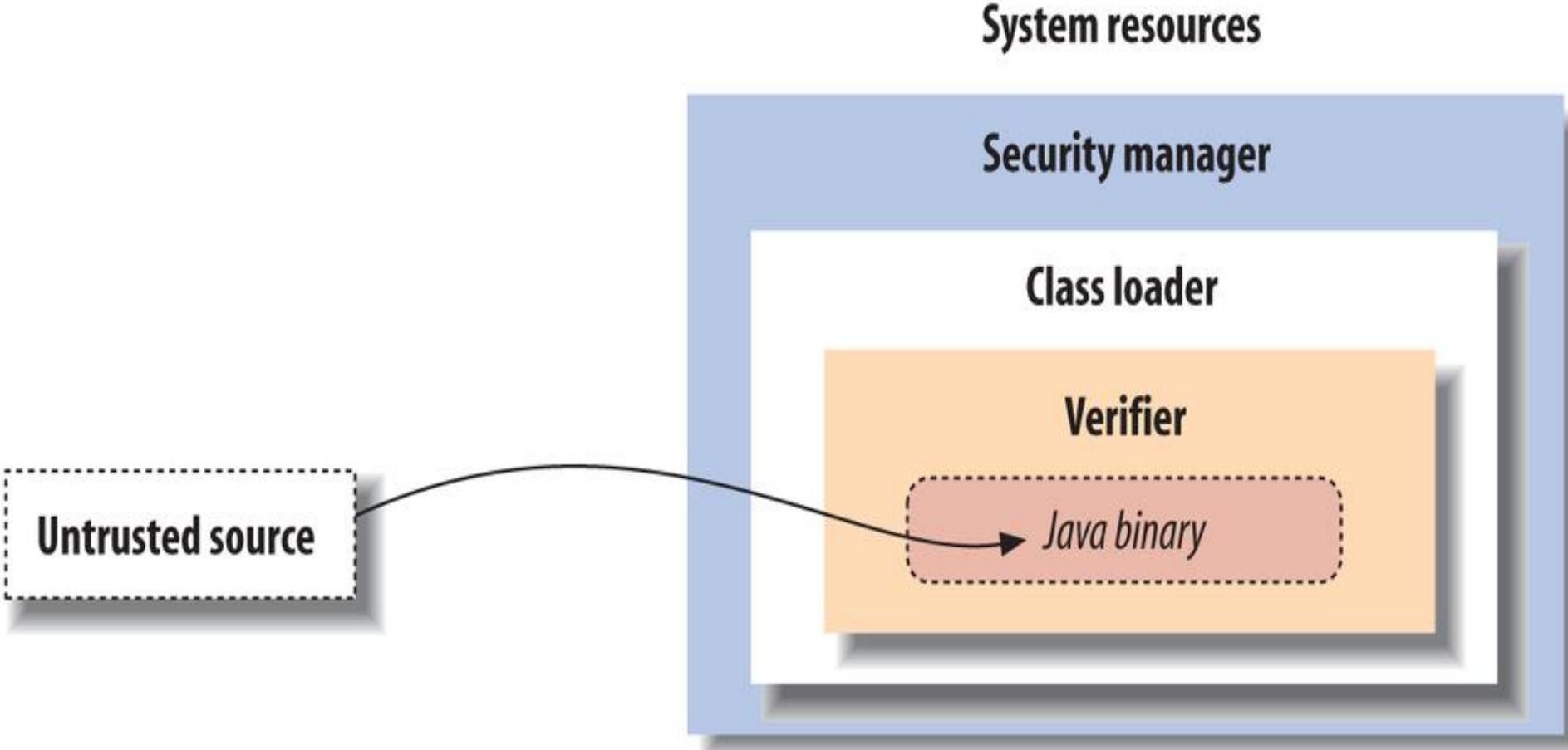
NOTA: differenza fra istanza e classe dello stub

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)



CARICAMENTO CODICE NON FIDATO



SICUREZZA E SECURITY MANAGER IN RMI

In ogni JVM si possono avere molti **Classloader**

Per ogni **ClassLoader**, può essere attivato un **Security Manager** ([e dovrebbe esserlo](#)), ossia un garante e controllore di correttezza e sicurezza di ogni singola operazione per quella località prima che sia eseguita

Per specificare cosa autorizzare, lo si definisce in un file di **policy di autorizzazioni**

Sia il client che il server devono essere lanciati specificando il [file con le autorizzazioni \(file di policy\)](#) consultato dal **security manager** ([per il controllo dinamico della sicurezza](#))

SICUREZZA E SECURITY MANAGER IN RMI

Per l'esecuzione sicura del codice si richiede l'utilizzo del **RMSecurityManager**

- RMSecurityManager 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
RMSecurityManager()); }
```

Esempio di **invocazione con la indicazione del file di policy**

- **C:** `java -Djava.security.policy=echo.policy`
`EchoRMIClient`
- **Server:** `java -Djava.security.policy=echo.policy`
`EchoRMIServer`

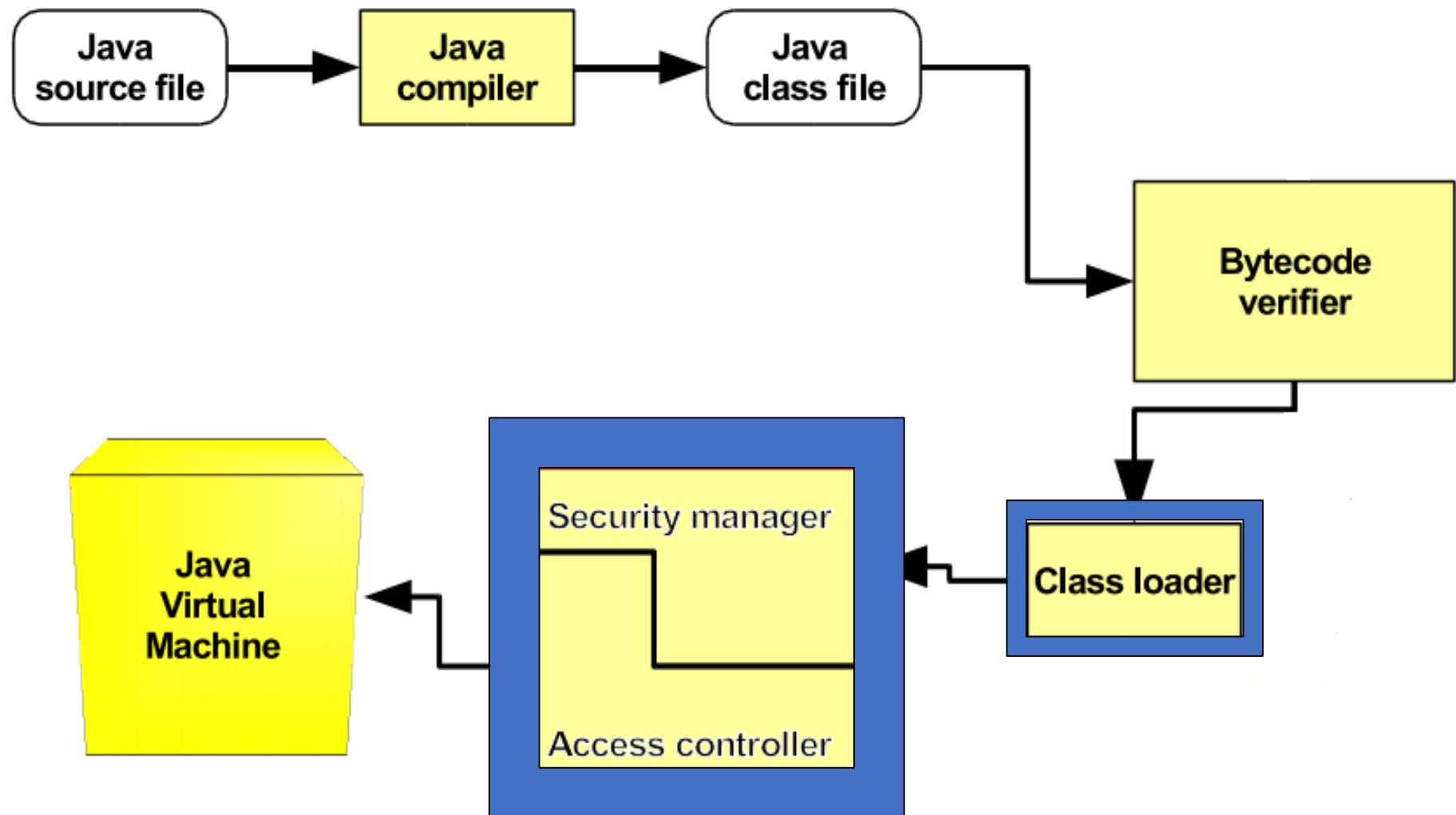
FILE DI POLICY PER SPECIFICARE LA SICUREZZA

Struttura del file di policy: **sequenza delle operazioni consentite**

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

- Il primo permesso consente al client e al server di **instaurare le connessioni necessarie all'interazione remota** (porte utente)
- Il secondo permesso consente di **prelevare il codice** da un **server http e relativa porta 80**
- Il terzo permesso consente di **prelevare codice** a partire dalla **radice dei direttori consentiti**

CATENA DINAMICA JAVA



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() + rebinding()`)

Soluzione:

i metodi bind(), rebinding() 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. sull'host in cui vengono effettuate le chiamate al registry deve esserci sempre **almeno un registry già in esecuzione**

RMI REGISTRY: IL PROBLEMA DEL BOOTSTRAP

Come avviene l'avvio del sistema (**bootstrap**) ed è possibile 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

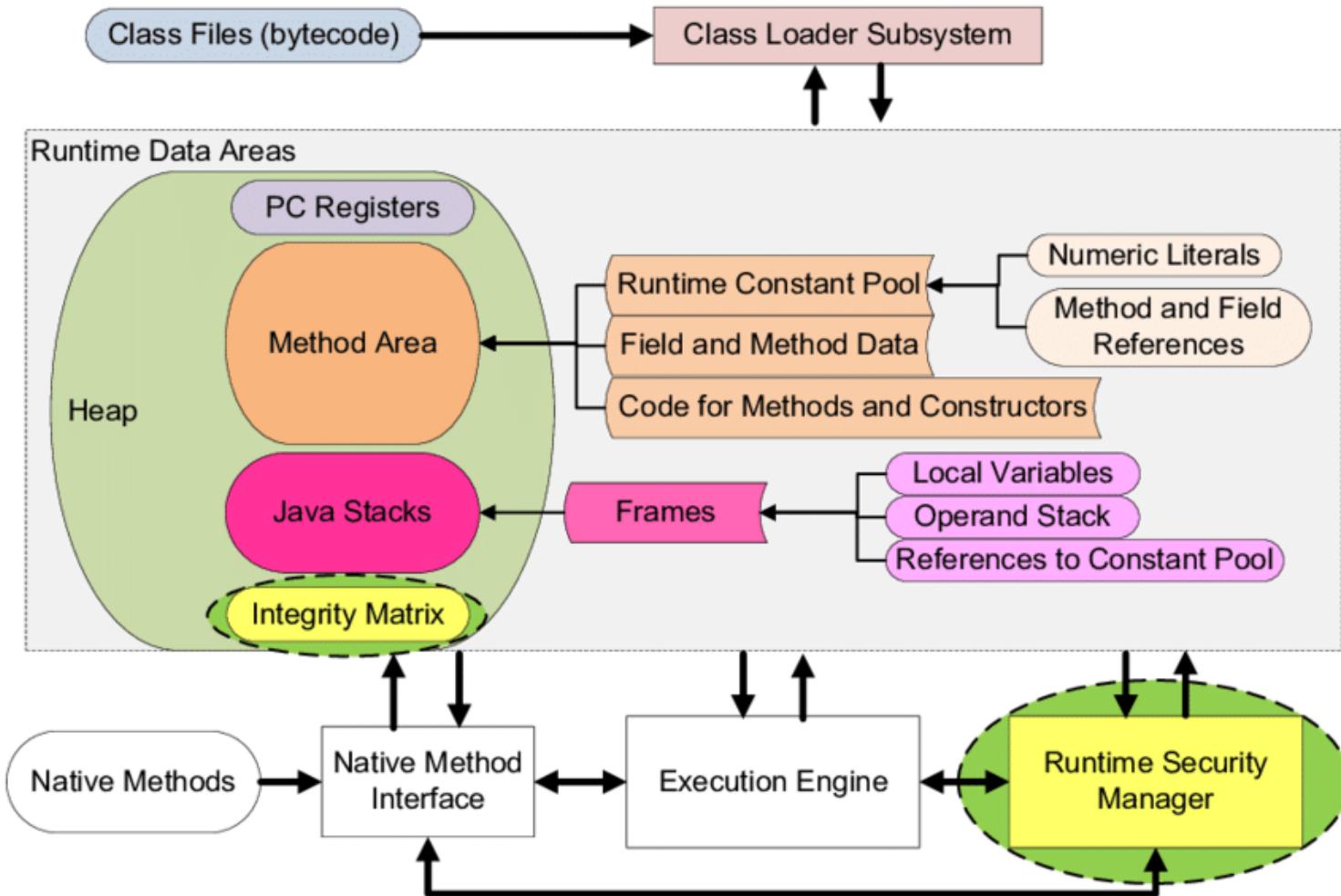
Come ottenere un'istanza dello stub del registry senza consultare un registry?

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**

VISIONE INTERNA MEMORIA JAVA

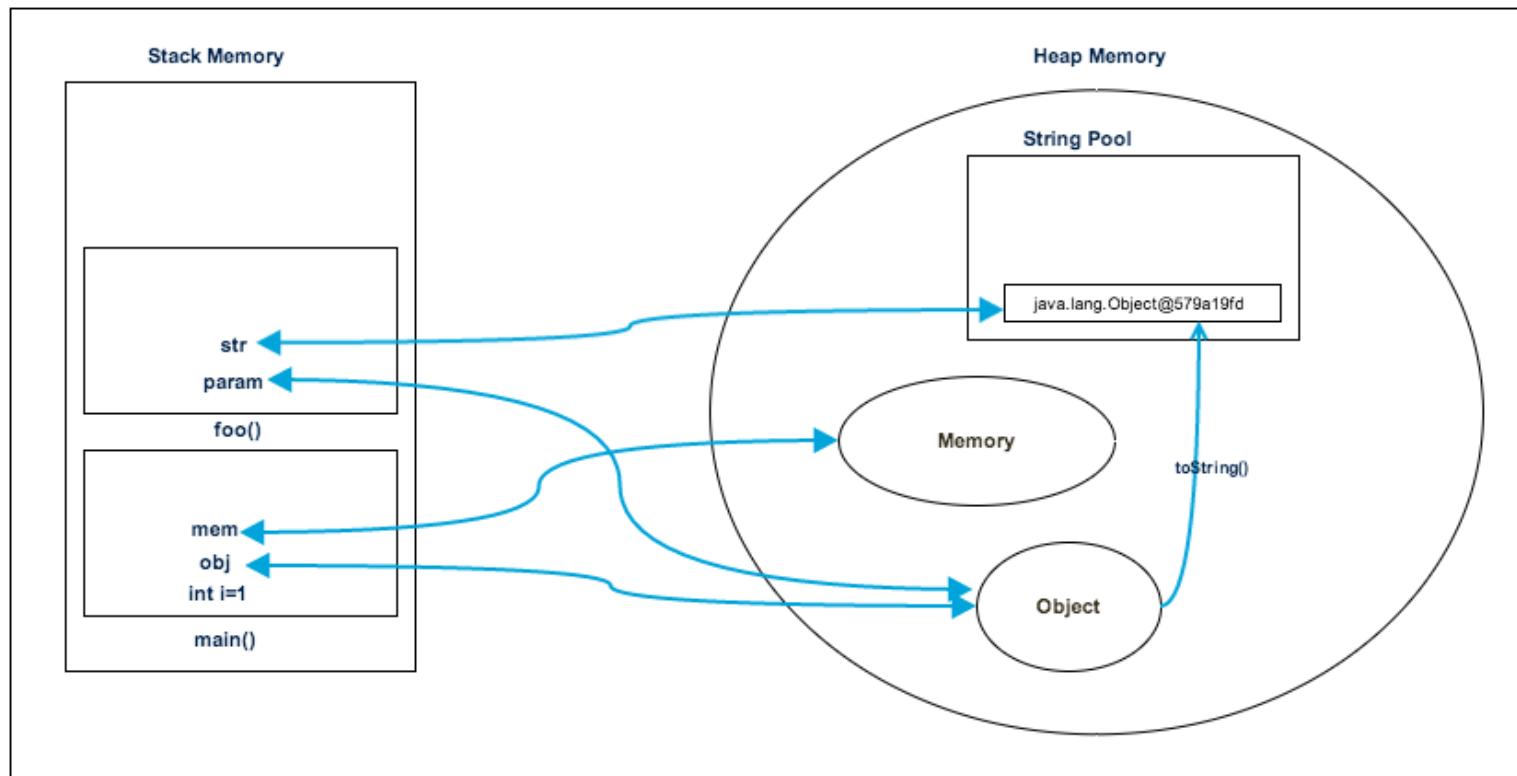
La memoria in Java è tipicamente caricata **by need** e non in modo statico preventivo: le istanze e le classi sono caricate solo quando sono effettivamente necessarie alla esecuzione



VISIONE DELLA MEMORIA

La memoria in Java si basa su Heap nella cui memoria si mettono tutti gli oggetti (istanze e classi) allocati dinamicamente

I singoli stack (uno per ogni thread) riportano le copie degli oggetti (con le variabili istanza e interfaccia interne che tipicamente puntano agli oggetti nell'heap)



GESTIONE MEMORIA HEAP

In un heap gli oggetti richiedono una strategia di allocazione e di deallocazione

Due strategie:

- **Reference counting**

Ogni oggetto mantiene un contatore interno che conta i riferimenti che ha distribuito (correttezza del valore?)

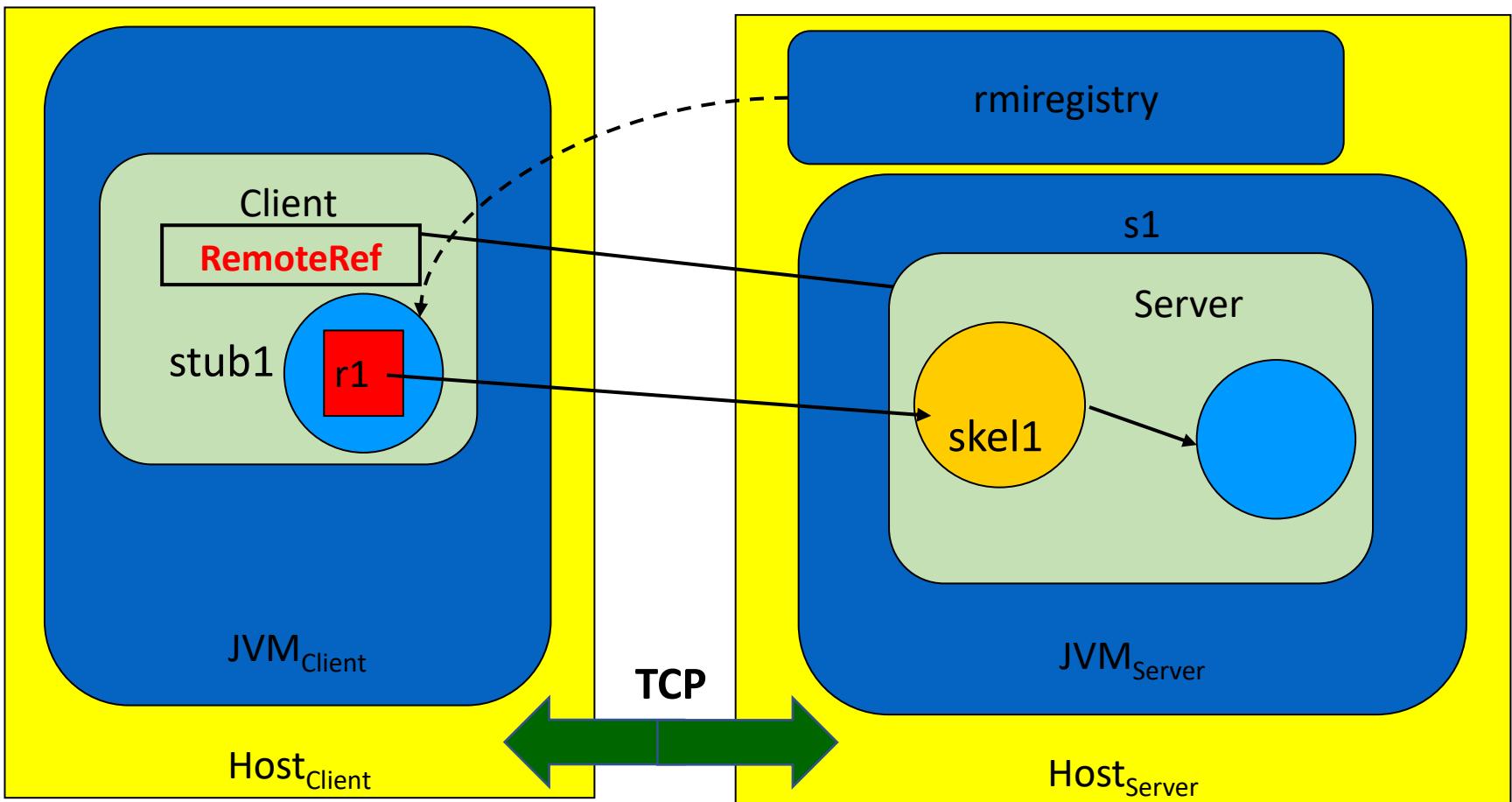
- **Garbage collection**

Una attività procede a deallocare gli oggetti che non sono riferiti da nessun altro oggetto (le classi sono trattate come gli oggetti)

In Java locale si lavora con **Garbage Collector** (ispirato al modello di **Djikstra** *mark&sweep della memoria on the fly*, mentre il sistema esegue)

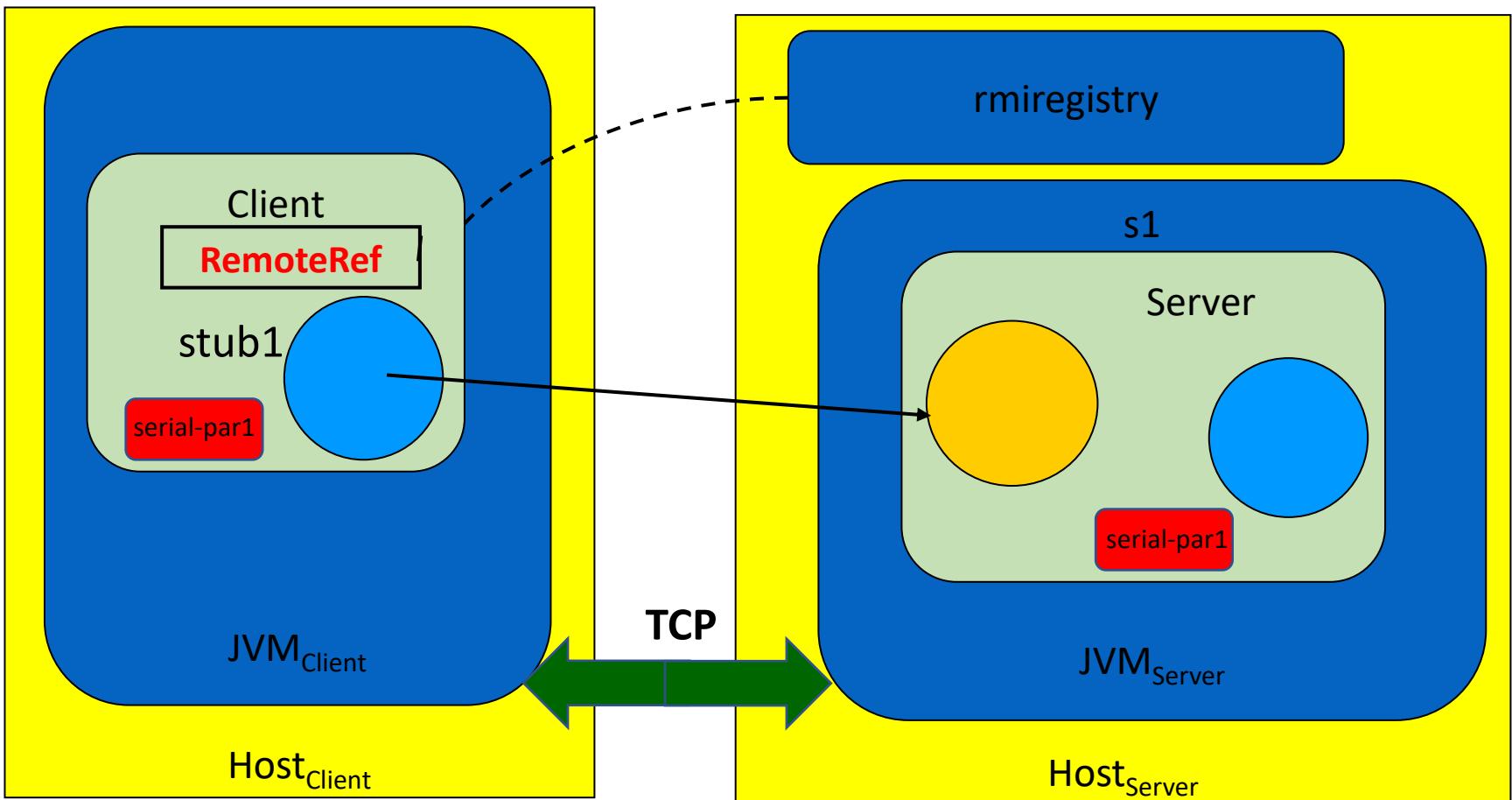
In RMI abbiamo una **strategia distribuita basata sul reference counting**: il numero di reference si ottiene con query dai server ai potenziali clienti

VISIONE DI INSIEME



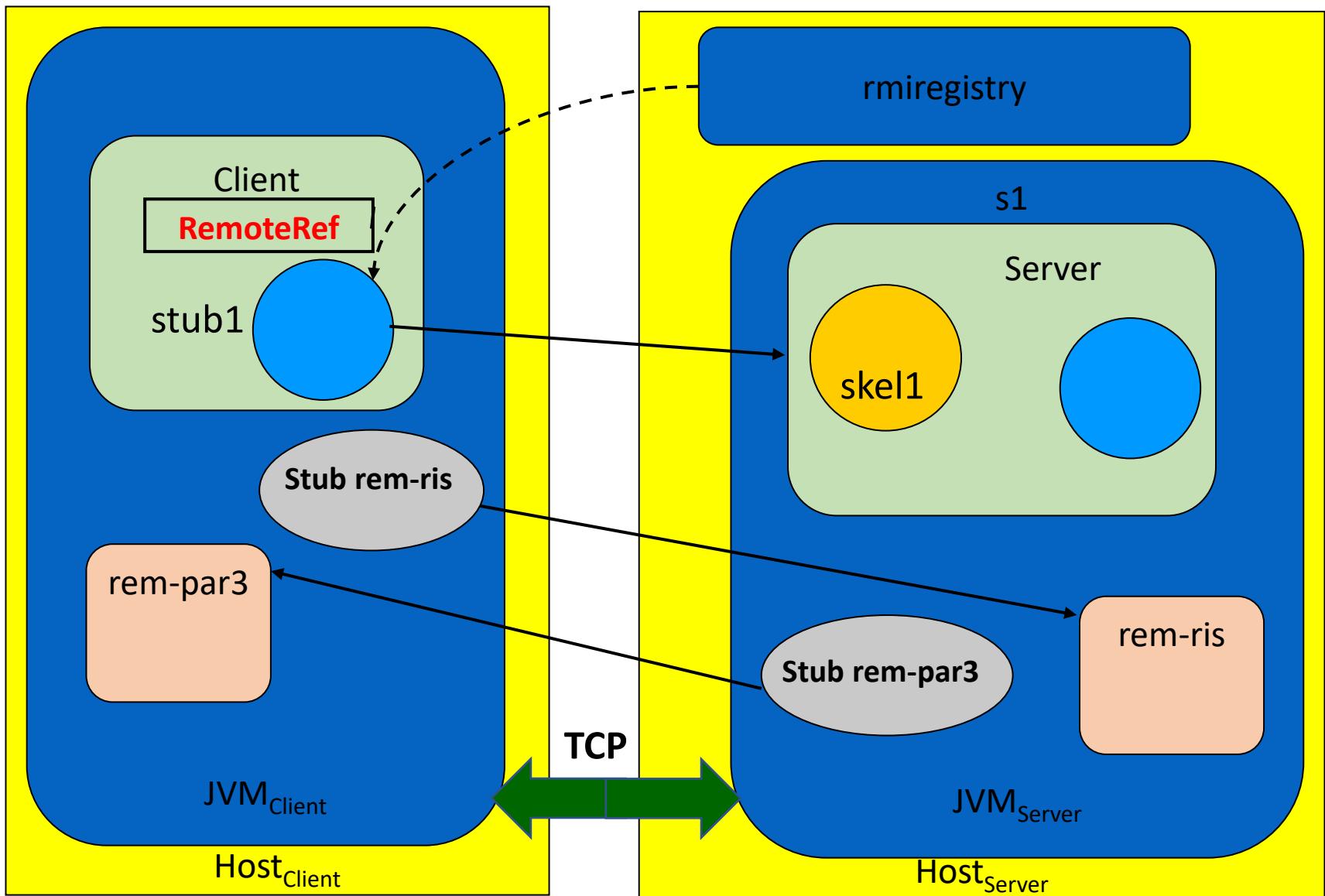
Il tipico metodo remoto invocato in modo sincrono bloccante
ris = vIntRem. metodo (par1, par2, par3);
prevede parametri di tipo diverso e trattati diversamente

VISIONE DI INSIEME



Il tipico metodo remoto invocato in modo sincrono bloccante
rem-ris = vIntRem. metodo (serial-par1, par2, rem-par3);
prevede parametri anche remoti (sia rem-ris sia rem-par3)

REM-RIS = VINTREM. METODO (SERIAL-PAR1, PAR2, REM-PAR3);



REMOTE REFERENCES

Il sistema di nomi RMI-registry è uno strumento di default per la propagazione della visibilità globale

Lo usiamo per ottenere il primo riferimento remoto del servitore in una applicazione C/S

Poi i riferimenti remoti si possono propagare attraverso le chiamate ai metodi remoti e con parametri opportuni

In generale, RMI non si propone come uno **strumento in-the-large** per una serie di ragioni

- Il registry registra solo i **servizi locali** (NON sistema di nomi globale)
- Il GC è molto **complesso e pesante** come costo
- Il costo delle **RMI è alto e lo strumento poco flessibile**
- RMI presentano meccanismi **non estendibili** e strategie **non flessibili**

BIBLIOGRAFIA

Sito della Sun: <http://java.sun.com/products/jdk/rmi/>

Oracle <http://www.oracle.com/technetwork/java/overview-139128.html>

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

PATH ED ESECUZIONE

I **processi nella esecuzione** devono poter accedere ai componenti di programmi necessari nell'esecuzione

Bash: usa il **set di PATH (e CLASSPATH)**

Inserire nella propria HOME, il file ".profile" (creandolo se non esiste) per aggiungere il direttorio corrente a PATH e CLASSPATH; il file deve contenere le seguenti linee:

PATH=. :\$PATH

CLASSPATH=. :\$CLASSPATH

export PATH

export CLASSPATH

Si vedano inoltre le regole per la definizione di variabili di ambiente su shell di unix

ATTENZIONE: *il nome del file di profilo dell'utente cambia a seconda della distribuzione, ad esempio, in RedHat il file si deve nominare ".profile", mentre in altri (controlla distribuzione disponibile nei laboratori) il file deve essere nominato ".bashrc".*

PATH ED ESECUZIONE

RMIregistry, server e client devono poter accedere alle classi necessarie per l'esecuzione



Attenzione al directory dove vengono lanciati il registry, il server e il client

Ipotizzando di avere tutti i file .class nel directory corrente ("."), e di lanciare registry, client, e server dal directory corrente, **bisogna aggiungere al CLASSPATH tale directory**

Sotto Linux: aggiungere nella propria directory **HOME** il file "**.profile**" (creandolo se non esiste).

Il file di configurazione dell'utente **.profile** (**.bashrc**, ...) **deve** contenere le seguenti linee per aggiungere il directory corrente al CLASSPATH:

- **CLASSPATH=.:\${CLASSPATH}**
- **export CLASSPATH**

Questa è la **modalità standard in Linux per aggiungere o modificare una variabile di ambiente** (vedi caso della variabile di ambiente PATH nelle FAQ del corso)