Università degli Studi di Bologna
Facoltà di Ingegneria

Corso di

Reti di Calcolatori M

Chiamate di Procedura Remota: il caso Java RMI

Luca Foschini

Anno accademico 2014/2015

Agenda

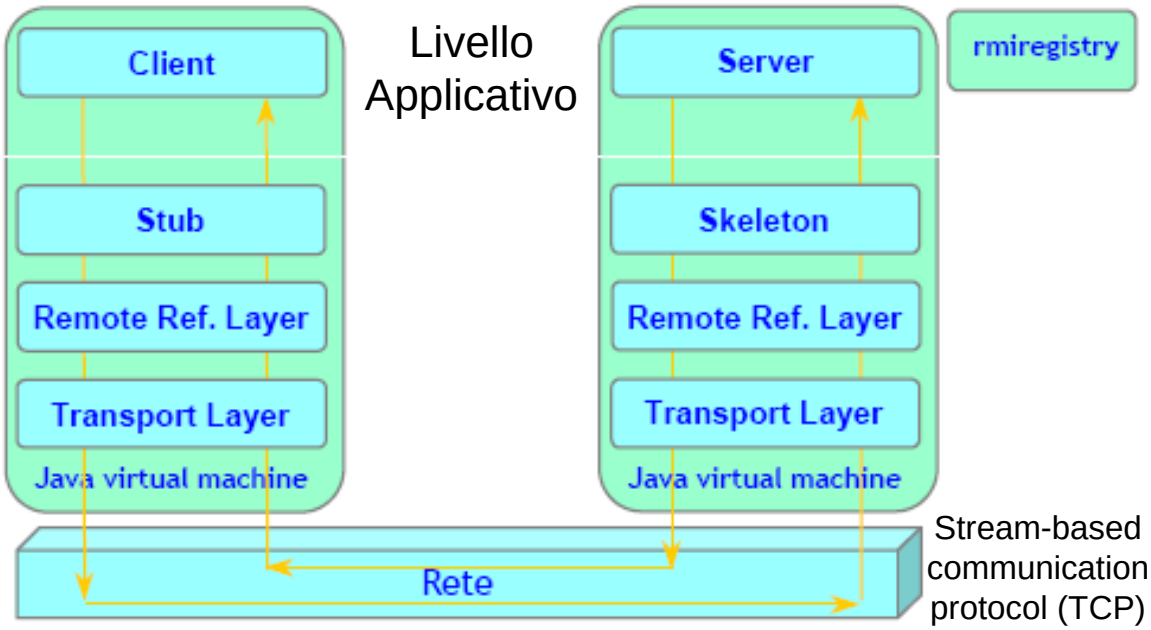
- Ripresa di RMI
- Architettura a livelli
- Generazione di Stub e Skeleton
- Passaggio dei parametri
- Modello di concorrenza
- Comunicazione
- Deployment
- Garbage collection distribuito

Ripresa RMI

- Modello solo per 1 solo linguaggio (Java)
- Non flessibile
  - Server **parallelo**
  - Chiamate **sincrone**
- Solo supporto a RPC
- Pochi altri servizi di supporto
  - Servizio di nomi (**rmiregistry**) molto **semplice** e **locale**
- Decisioni statiche

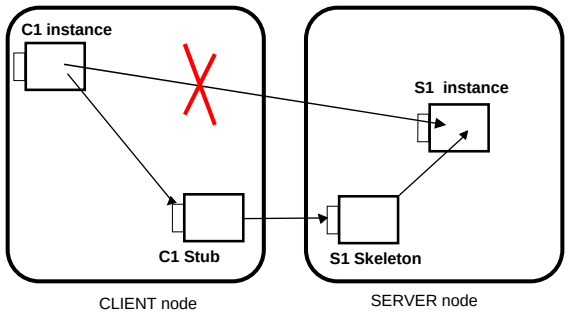
Architettura

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



Architettura a strati (1)

- Organizzazione a livelli:
  - Ogni livello realizza specifiche funzionalità
  - Separazione delle funzionalità e diversi livelli di astrazione
- Lo strato Stub e Skeleton
  - 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 → idea forte di infrastruttura attiva e distribuita

Architettura a strati (2)

Come **livelli di supporto** e **parte integrante della JVM**

- Lo strato Remote Reference Layer (RRL):
  - Responsabile della gestione del riferimento all'oggetto remoto
  - Scambio dati con il livello di trasporto
    - Astrazione 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 il parallelismo
  - Utilizzo di un protocollo proprietario
  - Possibilità di utilizzare protocolli applicativi diversi, purché siano connection oriented
    - E.g.: HTTP per poter invocare oggetti remoti che sono in esecuzione su macchine protette da firewall

Interfacce e Implementazione

- Separazione tra
  - definizione del comportamento → interfacce
  - implementazione del comportamento → classi

- Realizzare componenti remoti:
  - **Definizione** del comportamento, interfaccia che
    - Estende java.rmi.Remote
    - Propaga java.rmi.RemoteException
  - **Implementazione** comportamento, classe che
    - Implementa l'interfaccia definita
    - Estende java.rmi.UnicastRemoteObject

Riprendere in modo autonomo la realizzazione di C/S RMI da Reti di Calcolatori LA

Passaggio dei parametri

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

In **Locale:**
- **Copia → tipi primitivi**
- **Copia riferimento → tutti gli altri oggetti**

In **Remoto:**
- **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
  - **Marshaling**: processo di codifica degli argomenti e dei risultati per la trasmissione
  - **Unmarshaling**: processo inverso di decodifica di argomenti e risultati ricevuti

- In Java, grazie all'uso del BYTECODE, **NON** c'è un/marshaling, ma i dati vengono semplicemente serializzati 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

E i parametri passati per riferimento?

RMI Registry ancora

- Problema dell'avvio del sistema (**bootstrap**): per la partenza del sistema RMI è necessario contattare il primo oggetto remoto usato, cioè il Registry → *come fa il Client RMI ad ottenere il riferimento a lui?*
  - Ricordiamo che ogni oggetto remoto ha un **identificativo unico** all'interno della JVM nella quale è stato creato e registrato
  - Per contattare il Registry remoto, Java mette a disposizione la classe Naming i cui metodi hanno già tutte le informazioni necessarie per costruire (localmente) il riferimento al Registry di interesse:
    - **Host** e **porta** (passati come parametro nelle operazioni di lookup/rebind) su cui è in ascolto il processo in cui è stato creato e attivato il Registry
    - **Identificativo del Registry**: fissato a default come identificativo riservato dalla specifica RMI

- Il Registry presenta anche alcune **limitazioni**
  - NON è trasparente alla locazione
  - NON gestisce la sicurezza

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 allo stub
  5. lo stub:
     - effettua la de-serializzazione del valore di ritorno
     - restituisce il risultato al client

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 può avere implicazioni sulle invocazioni di oggetti remoti in esecuzione sulla JVM locale

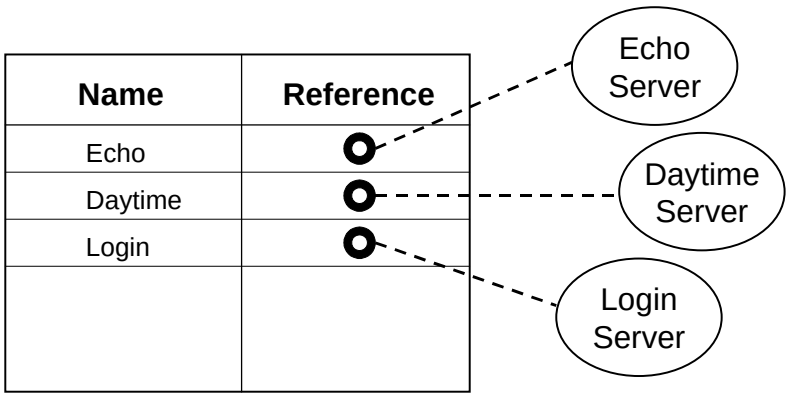
Livello di trasporto: la comunicazione

- 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

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

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



Implementazione del Registry

- Il Registry è un server RMI con interfaccia: **`java.rmi.registry.Registry`**

```
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;
}
```

- L'interfaccia Registry consente di gestire in modo coordinato più RMI Registry, sempre sullo stesso host, anche organizzandoli in gerarchie e strutture diverse
- Uso della classe **`java.rmi.registry.LocateRegistry`** come *Factory* per la creazione di nuovi oggetti Registry (sull'host del server)

La classe Naming

- 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)
```

`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!!

- La classe Naming agisce come un particolare stub locale per accedere alle funzionalità degli RMI Registry
- Tale stub **non** è legato ad un Registry specifico, ma utilizza il parametro **name** per risolvere e collegarsi a tempo di esecuzione al Registry di interesse

Attivazione del Registry

Due possibilità:

1. Usare il programma rmiregistry di Sun, lanciato specificando o meno la porta:

   ```
   rmiregistry
   rmiregistry 10345
   ```

   – lanciato in un processo separato (rispetto a quello della JVM in cui sono in esecuzione gli RMI Server)
   – struttura e comportamento standard

2. Creare all'interno del codice un proprio registry attraverso il metodo

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

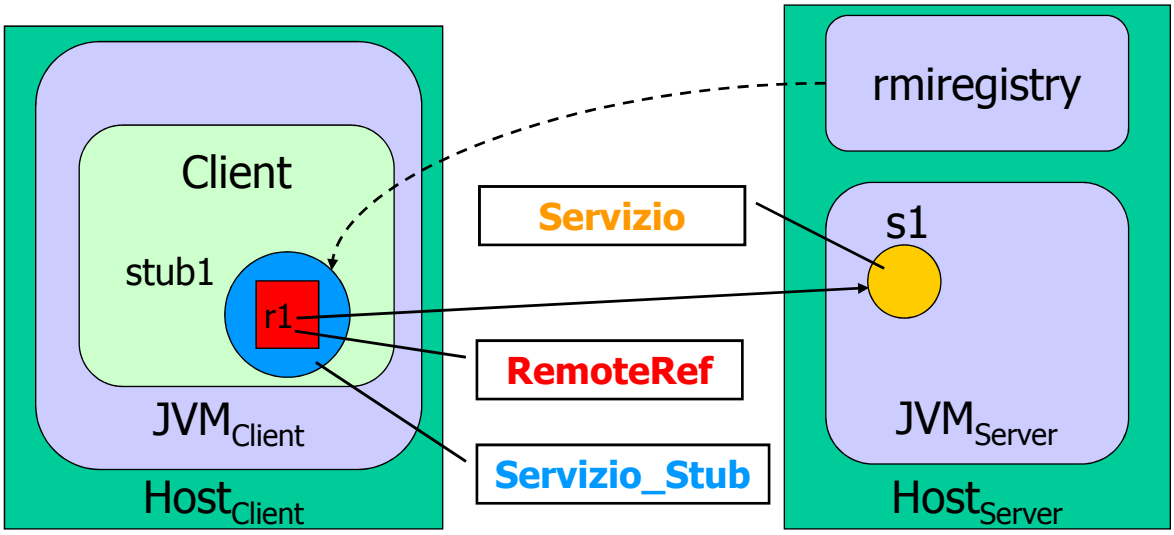
   – stessa istanza della JVM
   – struttura e comportamento personalizzabile (es. organizzazione gerarchica)

Stub e Skeleton: alcune considerazioni

- Le classi di stub e skeleton sono generate a partire dall'implementazione del server → nome della classe del server risolto a tempo di compilazione!
- Questo è un **limite** della soluzione RMI: in generale stub e skeleton potrebbero essere generati a partire dall'interfaccia (es. in CORBA)

- Per capire meglio… provare a guardare il sorgente di stub e skeleton
  – Provare a lanciare **`rmic`** con opzione **`-keep`**

Stub e Riferimenti Remoti



Il **Client** accede il **Server RMI** implementato dalla classe **Servizio** attraverso il riferimento allo stub locale stub1 (istanza della classe **Servizio_Stub** generata attraverso rmic) che contiene al suo interno un **RemoteRef** (r1) che consente al remote reference layer di raggiungere il server

Stub (fino a specifica 1.1)

- Contiene al suo interno il RemoteRef (**`super.ref`** di classe **`java.rmi.server.RemoteRef`**) usato dal remote reference layer per raggiungere l'oggetto remoto
- Lo stub effettua l'invocazione, gestisce la de/serializzazione, e spedisce/riceve i dati appoggiandosi sul RRL

RemoteRef all'oggetto remoto

Hash calcolato sulla classe del server per verificare la compatibilità di stub e skeleton

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

Intero indicante l'operazione richiesta

```
// 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 (fino a specifica 1.1)

- 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**)

Nome della classe del server

```
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
```


Stub e Skeleton (specifica 1.2 e seguenti)

- Dalla JDK1.2, la realizzazione di stub e skeleton è stata semplificata passando informazioni ulteriori *(nome metodo e classi dei parametri di ingresso e dell'uscita)* da client a server ***(estensione del protocollo RMI esistente)***
  - La disponibilità di tali informazioni rende possibile l'uso di *tecniche di riflessione* per effettuare il dispatching (lato server) → per risolvere dinamicamente l'operazione richiesta;
  - Nuove API per facilitare la realizzazione di stub e skeleton generici
- **Stub**: nuovo metodo **`invoke()`** messo a disposizione dalla classe **`RemoteRef`**

  `public Object invoke(Remote stubObj, Method method, Object[] params, long hashOpnum) throws Exception`
- **Skeleton**: non c'è più bisogno di generare lo skeleton
  - Unico skeleton generico che **recupera dinamicamente** i nomi del metodo e delle classi dei parametri dalla richiesta in arrivo usando la riflessione ed effettua il dispatching verso il metodo richiesto → alto costo della riflessione!!

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

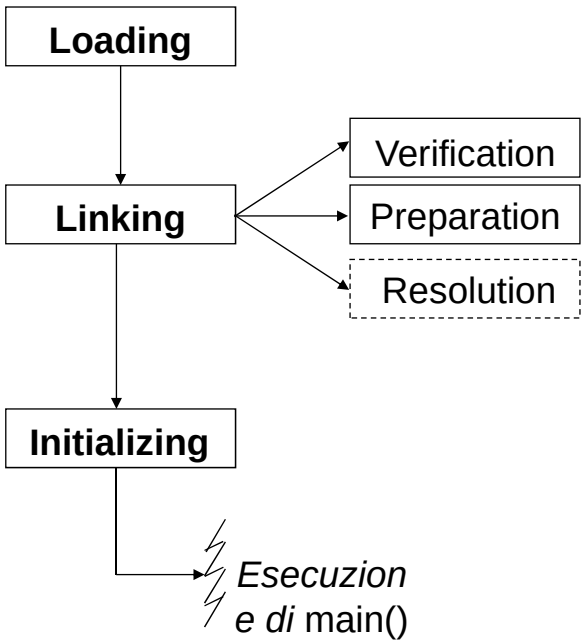
Java Class loading e JVM
(apriamo una parentesi)

Caricamento classi nella Java Virtual Machine, per passi:

- caricamento in memoria del file *.class*

- verifica byte-code
- allocazione spazio in memoria
- risoluzione di tutti i riferimenti simbolici ad altre classi

- inizializzazione di tutte le variabili statiche allocate



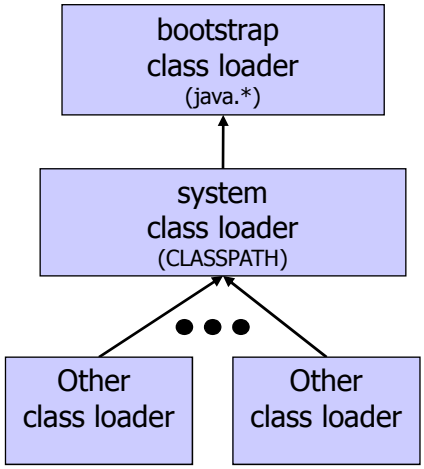
Java Class loading

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



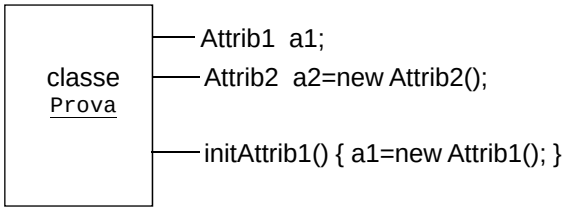
Il Class Loader di Java (1)

Caratteristiche:

- **"lazy" loading**: le classi sono caricate "on demand" e possibilmente all'ultimo momento

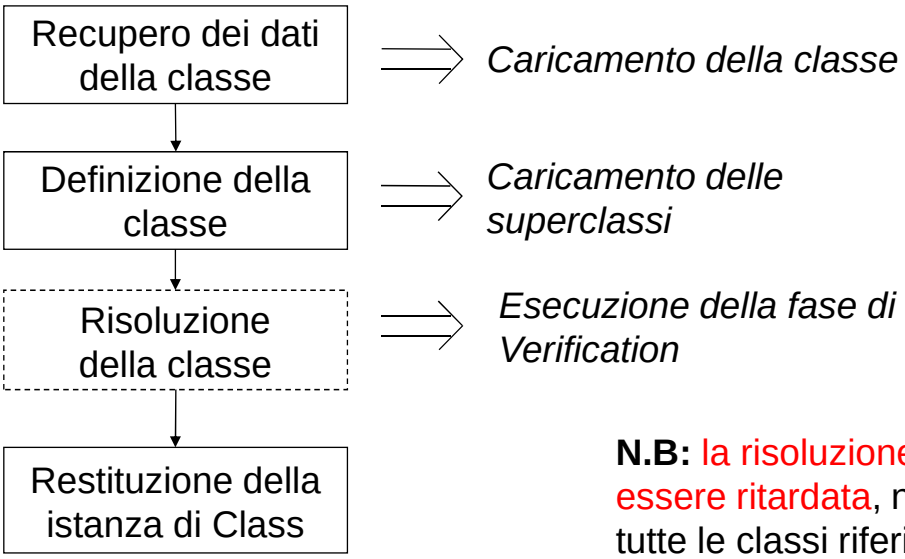
Esempio:

Prova p = new Prova();



- **programmabilità**
- **creazione di namespace multipli**

Il Class Loader di Java (2)

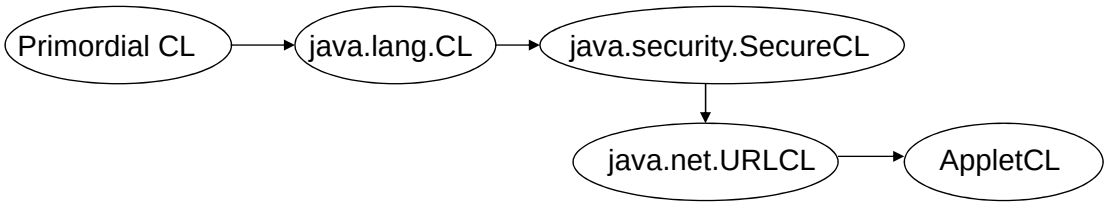


N.B: la risoluzione può essere ritardata, non tutte le classi riferite devono essere immediatamente caricate

Tipi di Class Loader

- **class loader interno:** solo per caricare le classi di Java (JDK 1.2)
- **secure class loader:** permette di implementare politiche di sicurezza più granulari
- **applet class loader:** ogni browser implementa un proprio class loader
- **URL class loader:** per caricare classi da un insieme di URL distinte

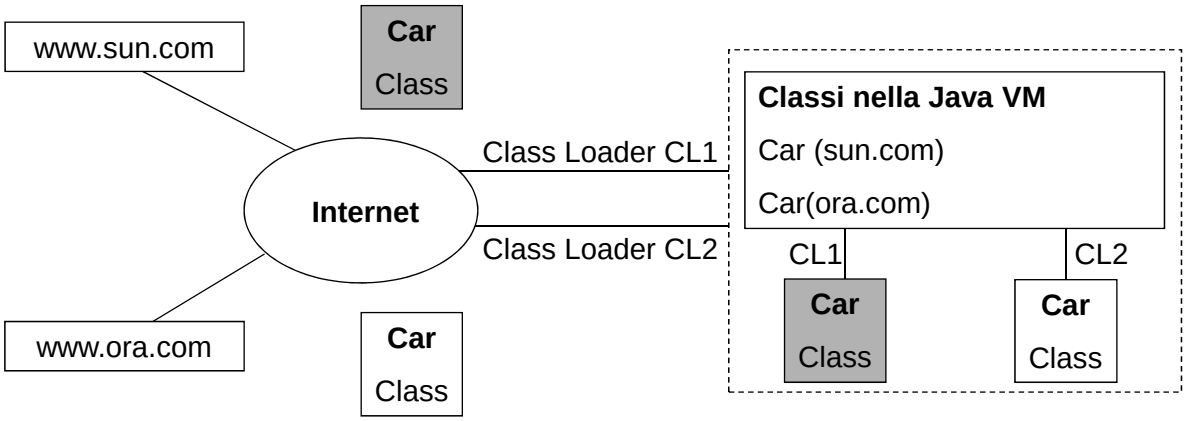
I programmatori possono definire **class loader (CL) propri** purché tutti estendano i tipi predefiniti



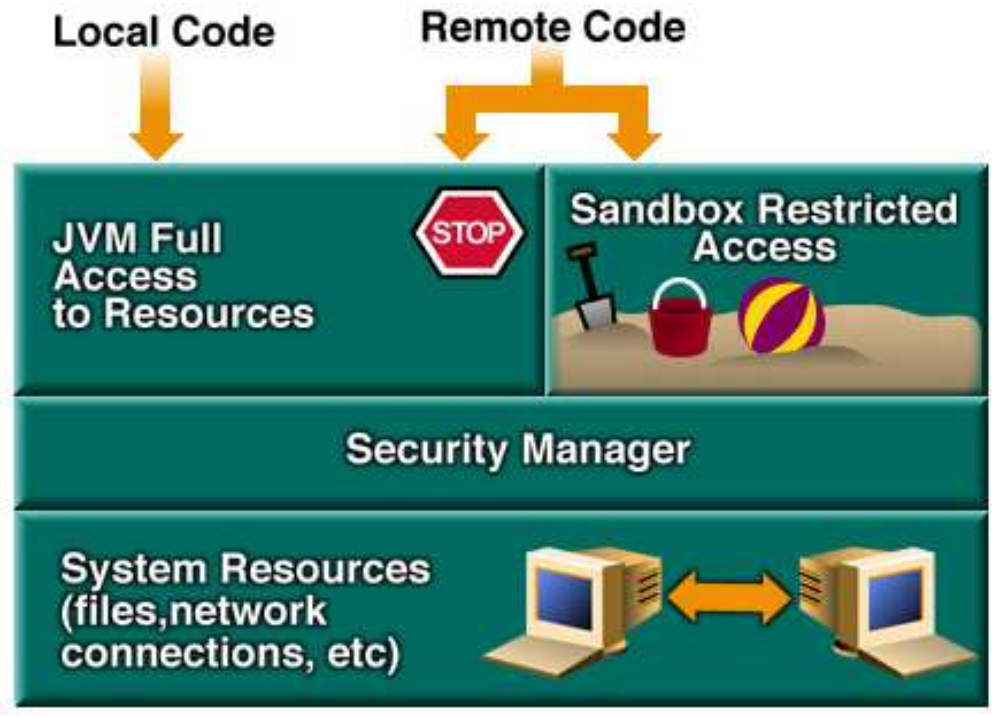
Class Loader e Sicurezza

Il Class Loader gioca un ruolo fondamentale

- crea **spazi di nomi separati**
- si coordina con il componente Security Manager che implementa le politiche di sicurezza



Modello di Sicurezza in JDK™ 1.0



Modello di Sicurezza in JDK™ 1.0

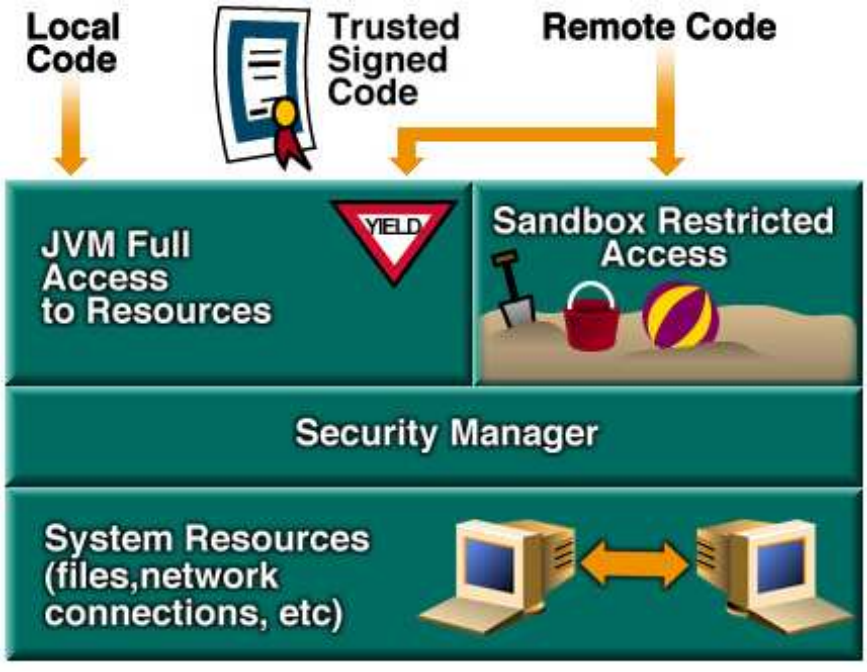
Vantaggi:

- la **sandbox** protegge l'accesso a tutte le risorse di sistema
- i programmatori di applicazioni (non di applet) **possono scrivere** un proprio Security Manager per "aprire" la sandbox

Limitazioni:

- modello **troppo restrittivo**
- i programmatori di applicazioni (non di applet) **devono scrivere** un proprio Security Manager per "aprire" la sandbox
- ad ogni politica diversa corrisponde **una nuova versione del SecurityManager** (vincolo molto forte)

Modello di Sicurezza in JDK™ 1.1



Modello di Sicurezza in JDK™ 1.1

Vantaggi:

- la **firma del codice** assicura:
  - autenticazione
  - integrità

Limitazioni:

- applicazioni **locali non sono sottoposte ad alcun controllo**
- tutte le **classi** presenti **nel CLASSPATH** sono considerate **fidate**

Modello di Sicurezza in JDK™ 1.2

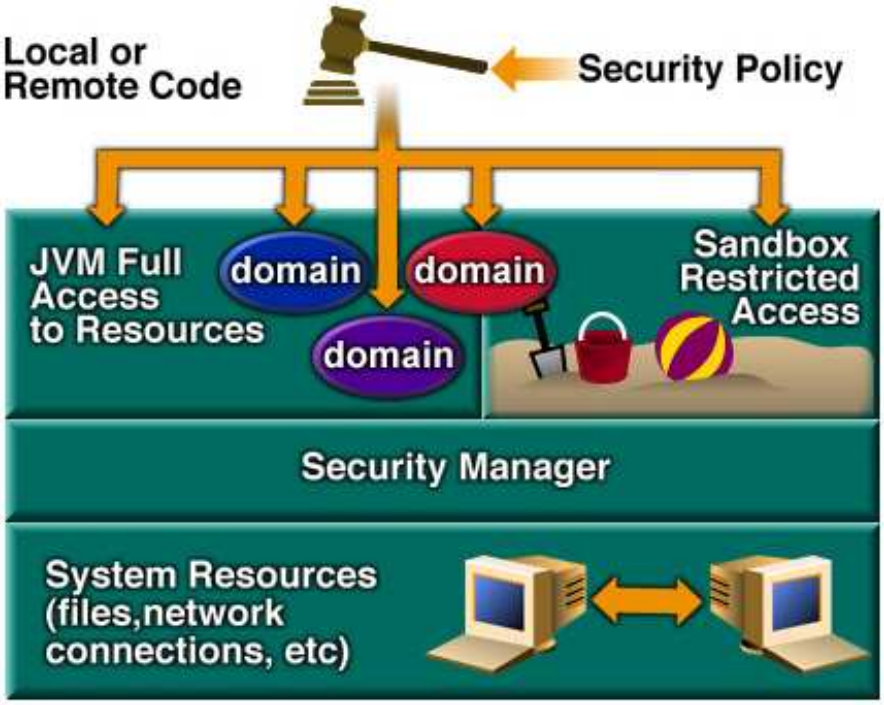
Obiettivi:

- estensione dei controlli di sicurezza sia alle applet sia alle applicazioni (esterne e locali)
- **configurabilità semplificata** delle politiche di sicurezza
- **controlli** di accesso **granulari**

Elementi caratterizzanti la nuova architettura:

- **politica** di **sicurezza**
- definizione di **permessi** **di accesso**
- **controllo** **d'accesso**

Modello di Sicurezza in JDK™ 1.2



Politica di Sicurezza (1)

Politica di sicurezza: matrice di controllo dell'accesso

Codice	Permessi
Li Gong applet, applet firmate	read, write /tmp and home/gong

Esempio di Politica di Sicurezza
(rappresentazione ASCII di default)

grant *signedBy* "*", *CodeBase* "http://java.sun.com/people/gong/"

```
{
    permission java.io.FilePermission "read, write", "tmp/*";
}
```

Politica di Sicurezza (2)

(chiudiamo la parentesi ☺)

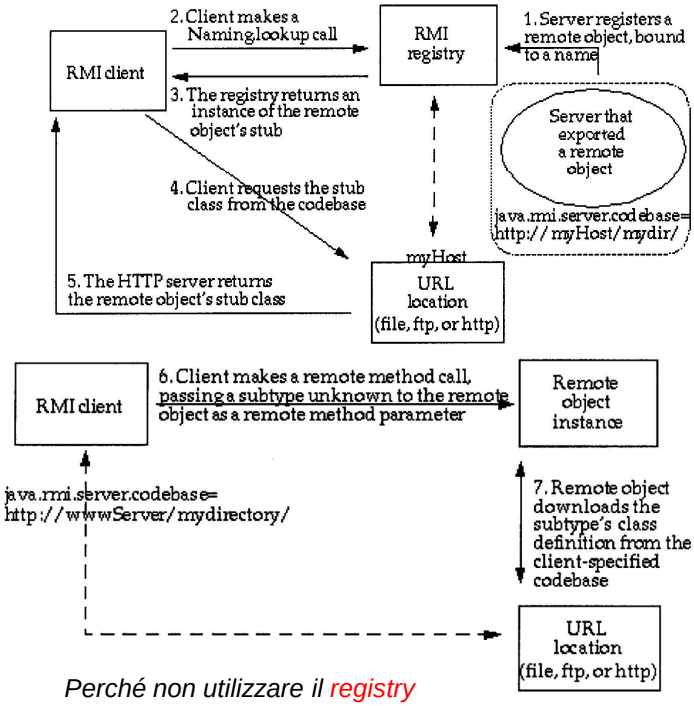
- **CodeSource** (chi/dove): località di provenienza del codice e insieme di certificati utilizzati per la firma del codice
- **Permissions:** permessi attribuiti al codice
- abstract class **Policy**:
  - public static Policy getPolicy()
  - public static void setPolicy(Policy policy)
  - public abstract Permissions getPermissions(CodeSource codesource)
  - public abstract void refresh()

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 servernecessari (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)

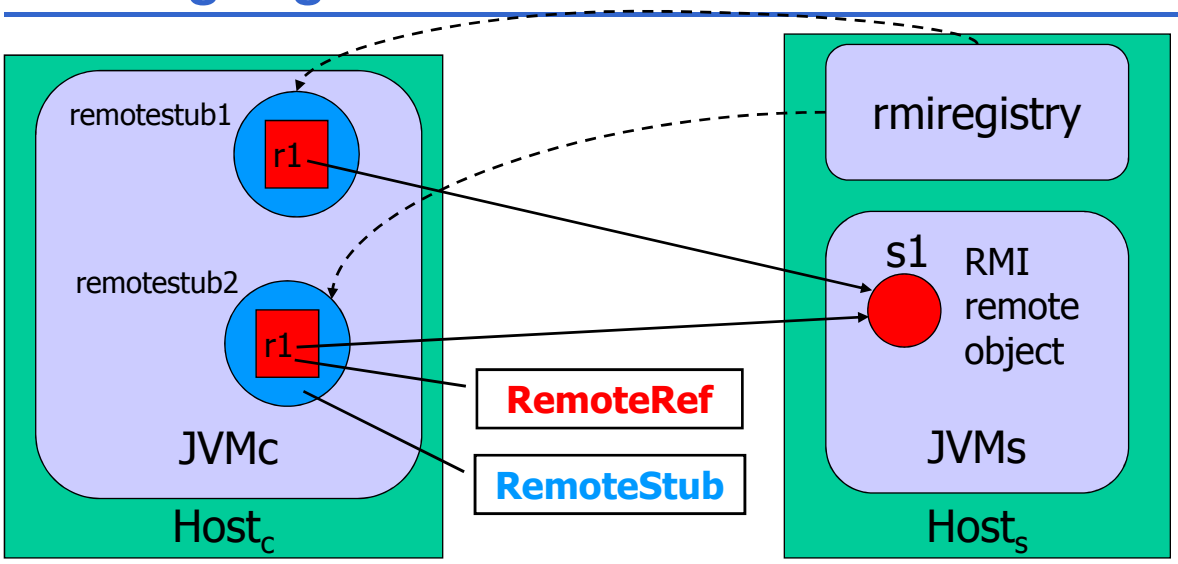


Perché non utilizzare il registry per passare direttamente i file class senza dover interpellare il web server?

Remote Reference Layer

- La principale funzionalità svolta è la gestione dei riferimenti agli oggetti remoti
- Come vengono trattati i riferimenti degli oggetti remoti per ciò che riguarda
  - Uguaglianza
  - Garbage collection
- In locale
  - Due oggetti vengono definiti uguali quando i riferimenti alle istanze di tali oggetti sono uguali, cioè hanno lo stesso identificatore
  - Un oggetto viene rimosso quando non esiste più alcun riferimento all'oggetto stesso
- E quando abbiamo a che fare con oggetti remoti?

Uguaglianza dei riferimenti remoti



remotestub1==remotestub2 **NO**

remotestub1.equals(remotestub2) **Sì**

Uguaglianza dei riferimenti remoti

- **In generale**, il metodo **`equals`** invocato sul ***RemoteStub*** verifica l'uguaglianza dei ***RemoteRef*** contenuti al loro interno, e viene mappato sul metodo **`remoteEquals`** realizzato dal ***RemoteRef***
  → ***Il confronto avviene IN LOCALE sull'host del client***

- **Nel caso specifico di `UnicastRef`** (riferimento utilizzato per gli **`UnicastRemoteObject`**) il metodo **`remoteEquals`** verifica
  - Endpoint (host:porta) di ascolto per le richieste RMI della JVM
  - Identificatore univoco (**`ObjID`**) generato per ogni oggetto esportato dalla JVM come **oggetto remoto RMI**

- **Nota**: problema del bootstrap un client deve riuscire a riferire il registry (l'oggetto remoto che lo implementa) pur non conoscendone l'identità
  → Soluzione: RMI stabilisce alcuni identificatori (ObjID fissi), ad esempio
    - **`REGISTRY_ID`** per l'rmiregistry
    - **`DGC_ID`** per il garbage collector distribuito

Riferimenti remoti e distruzione degli oggetti

Per uniformità con la gestione del ciclo di vita degli oggetti Java locali, RMI si pone il problema dell'eliminazione degli oggetti remoti

- Per realizzare ciò l'oggetto remoto deve conoscere in ogni istante **quanti stub lo stanno riferendo**
- **MA** gli ambienti distribuiti complicano il tutto (possibili **guasti di rete**, **caduta della macchina client**, …)

Scelta di progetto **molto forte** che
limita l'applicabilità di RMI ad **ambienti locali**
(i soli nei quali è possibile alto grado di coordinamento)

Distributed Garbage Collection (DGC)

- In un sistema distribuito si vuole avere la deallocazione automatica degli oggetti remoti che non sono più riferiti da nessun client.
  - Il sistema RMI utilizza un algoritmo di garbage collection basato sul conteggio dei riferimenti
  - Ogni JVM aggiorna una serie di contatori ciascuno associato ad un determinato oggetto
  - Ogni contatore rappresenta il numero dei riferimenti ad un certo oggetto che in quel momento sono attivi su una JVM
  - Ogni volta che viene creato un riferimento ad un oggetto remoto il relativo contatore viene incrementato. Per la prima occorrenza viene inviato un messaggio che avverte l'host del nuovo client
  - Quando un riferimento viene eliminato il relativo contatore viene decrementato. Se si tratta dell'ultima occorrenza un messaggio avverte il server

NOTA: prospettiva **molto coordinata** che ha senso
principalmente per sistemi locali di piccole dimensioni

DGC: il lease

Java RMI si pone il problema della caduta improvvisa di un client. Per risolvere tale problema DGC usa un meccanismo basato su **lease** (contratto)

- Il client detiene il contratto e DEVE rinnovarlo ad intervalli regolari (lease)
- Se il client non rinnova il lease, l'infrastruttura lato server assume che il client sia caduto o ci siano stati problemi di rete, quindi decrementa il contatore dei riferimenti remoti

Anche in questo caso abbiamo una soluzione ad **alto costo** (dal punto di vista dell'overhead di comunicazione) **NON attuabile** in ambienti **non locali** e **geograficamente molto distribuiti**

Oltre Java RMI: problemi aperti

- **Costo RMI**
  - Supporto pensato per **sistemi di piccole dimensioni** e **deployment locali** (es. DGC)
- **Sincronicità**
  - Ogni invocazione sull'oggetto remoto è di tipo **sincrono** e **bloccante**
  - Java **non** permette interazioni di tipo asincrono o non bloccanti a livello di supporto
- **Attivazione** e **ciclo di vita** degli oggetti remoti
  - **Non** c'è supporto all'attivazione dinamica degli oggetti remoti
  - Alcune politiche di attivazione supportate dalla JDK1.2, ma piuttosto **limitate** e **solitamente non utilizzate**
- **Eterogeneità** ambienti di sviluppo e linguaggi
  - **Non** è possibile utilizzare linguaggi diversi da Java
  - RMI IDL è Java, ossia il **linguaggio di implementazione**

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: lfoschini@deis.unibo.it
  - Home page: www.lia.deis.unibo.it/Staff/LucaFoschini