Il linguaggio Java

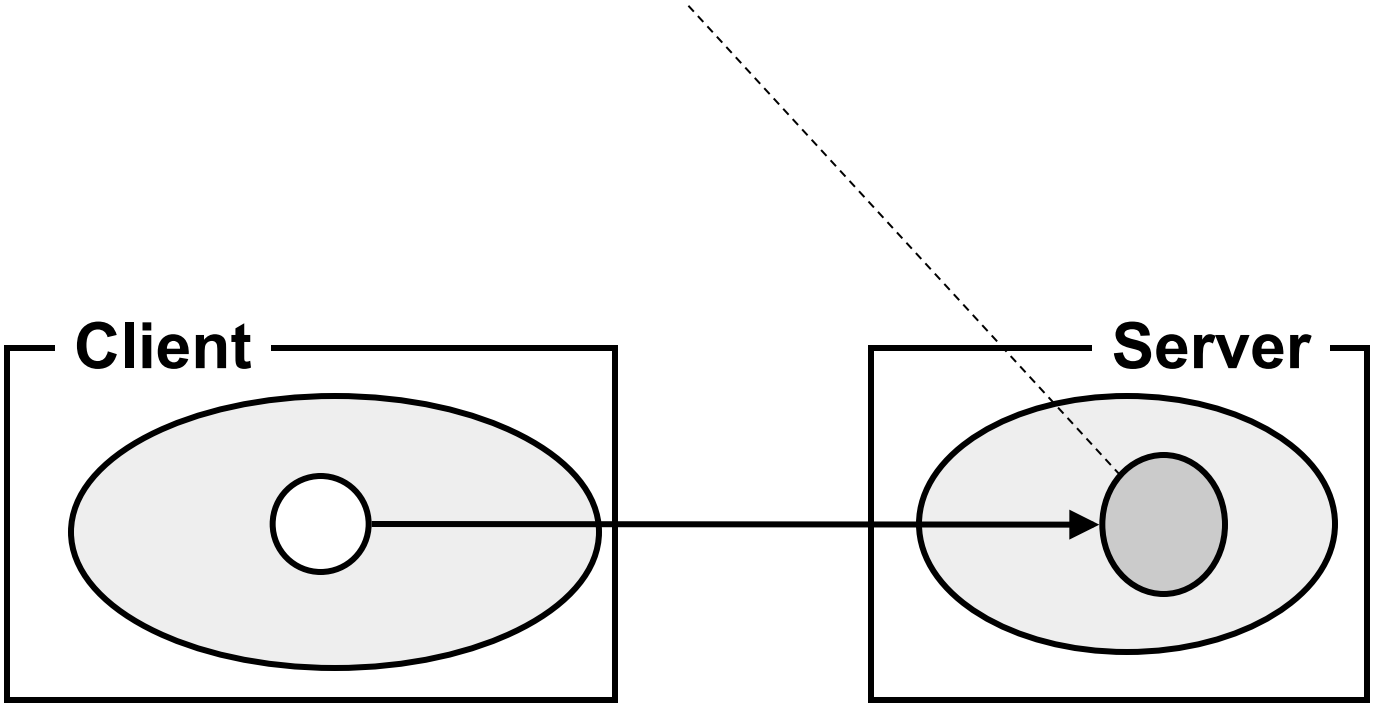
Remote Method Invocation (RMI)

RMI

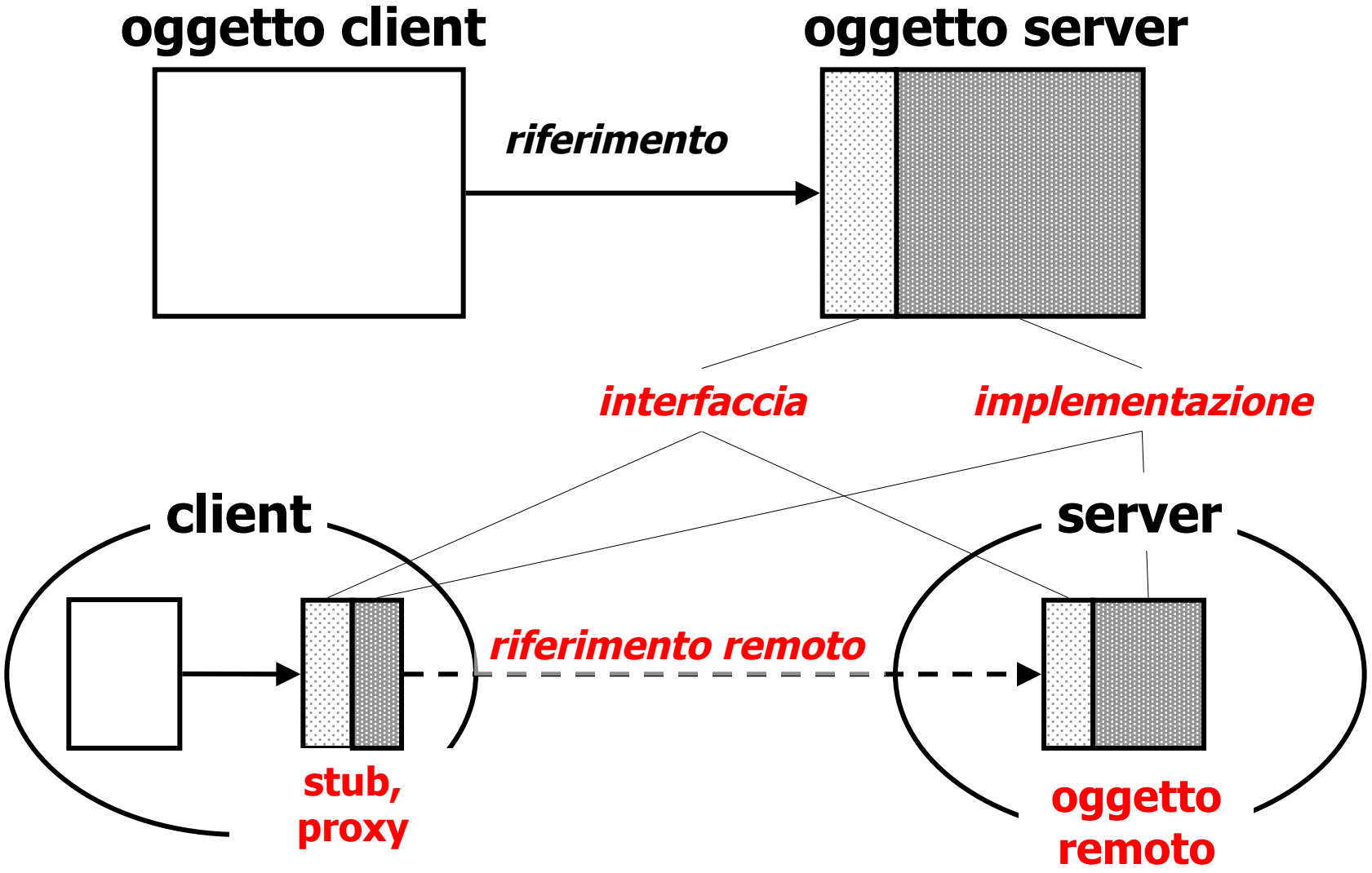
MECCANISMI BASE

Oggetto remoto

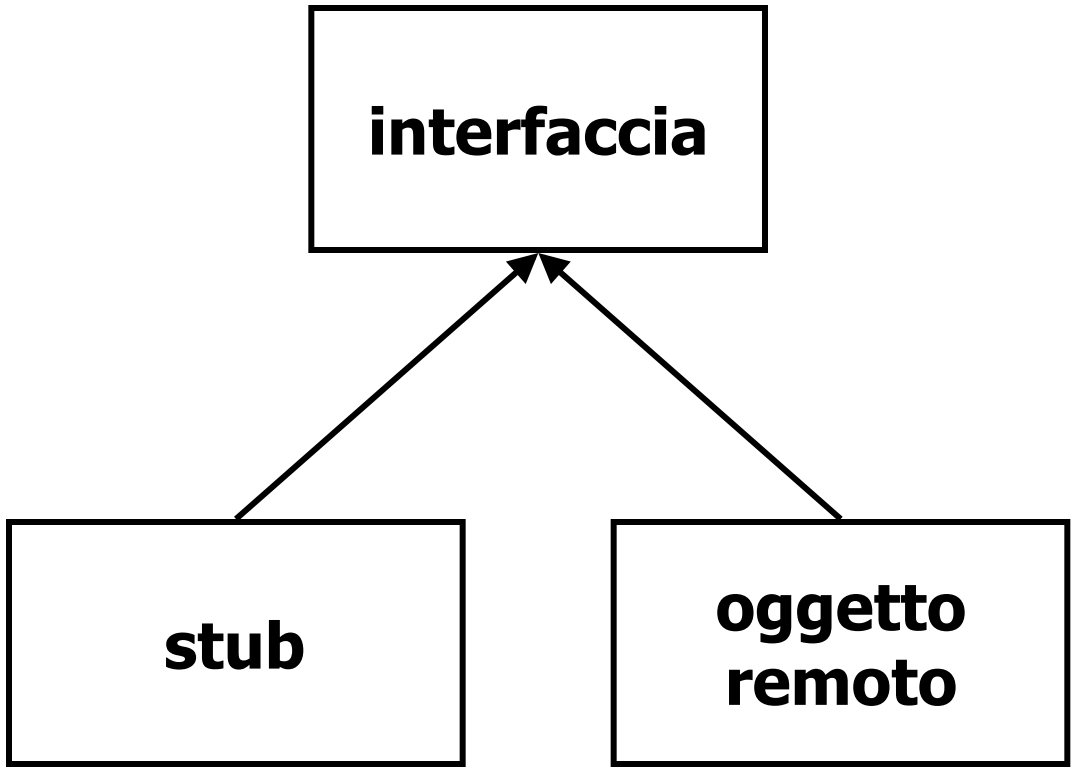
Oggetto remoto: oggetto i cui metodi possono essere invocati attraverso la rete

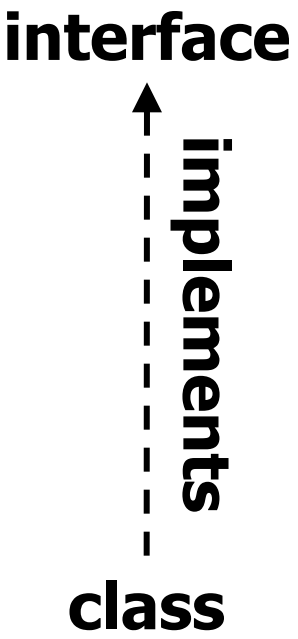


Schema di principio



Schema di principio





Esempio Calculator

Realizzazione di un oggetto remoto

- **Progettare l'interfaccia**
  - l'interfaccia estende **Remote**
  - la clausola **throws** di ciascun metodo remoto specifica **RemoteException** (eccezione controllata)
- **Realizzare l'interfaccia (oggetto remoto)**
  - Implementare ciascun metodo remoto
  - Definire il costruttore con una clausola **throws** che specifica **RemoteException**
  - Eventualmente dichiarare altri metodi, non specificati nell'interfaccia (invocabili solo localmente)

Realizzazione di un oggetto remoto

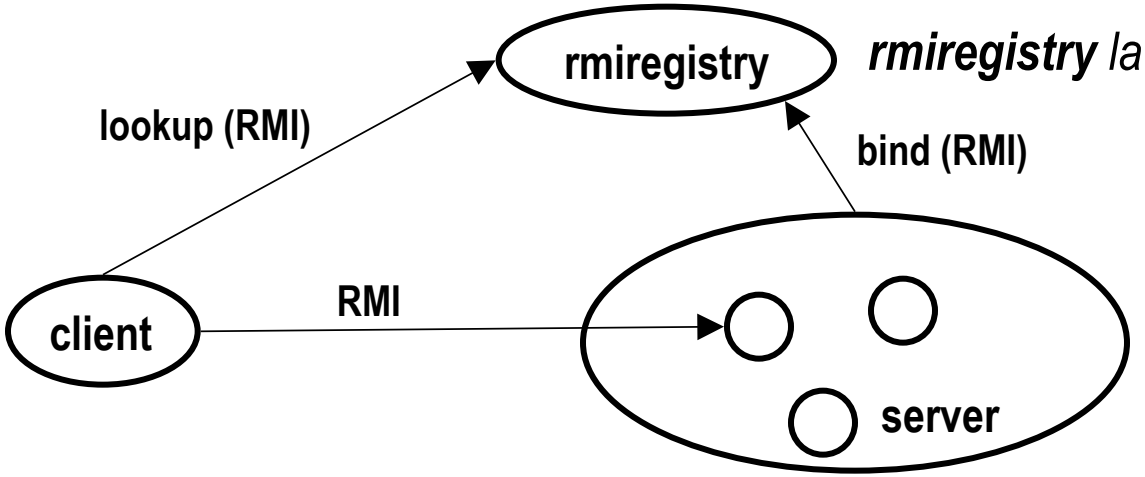
- **Realizzare il server**
  - Creare ed installare un Security Manager (quando serve)
  - Creare ed esportare uno o più oggetti remoti
  - Registrare gli oggetti remoti (quando serve)
- **Realizzare il cliente**
  - Creare ed installare un Security Manager (quando serve)
  - Localizzare gli oggetti ed ottenere gli stub relativi
  - Invocare i metodi remoti

Esportare e registrare un oggetto

Prima che l'oggetto remoto possa essere utilizzato, il server deve

- **esportare l'oggetto**
  - Il server deve rendere l'oggetto ***visibile*** in rete e ***disponibile*** a ricevere richieste remote
- **registrare l'oggetto**
  - Il server può associare un ***nome*** (stringa) ad un oggetto remoto per mezzo di un naming/directory service (operazione **bind**)
    - I clienti utilizzeranno tale nome per interrogare il naming/directory service ed ottenere un riferimento remoto per l'oggetto (operazione **lookup**)
    - Più precisamente, il server associa un nome al riferimento remoto dell'oggetto

Nominazione degli oggetti



rmiregistry *lavora sulla porta* ***1099***

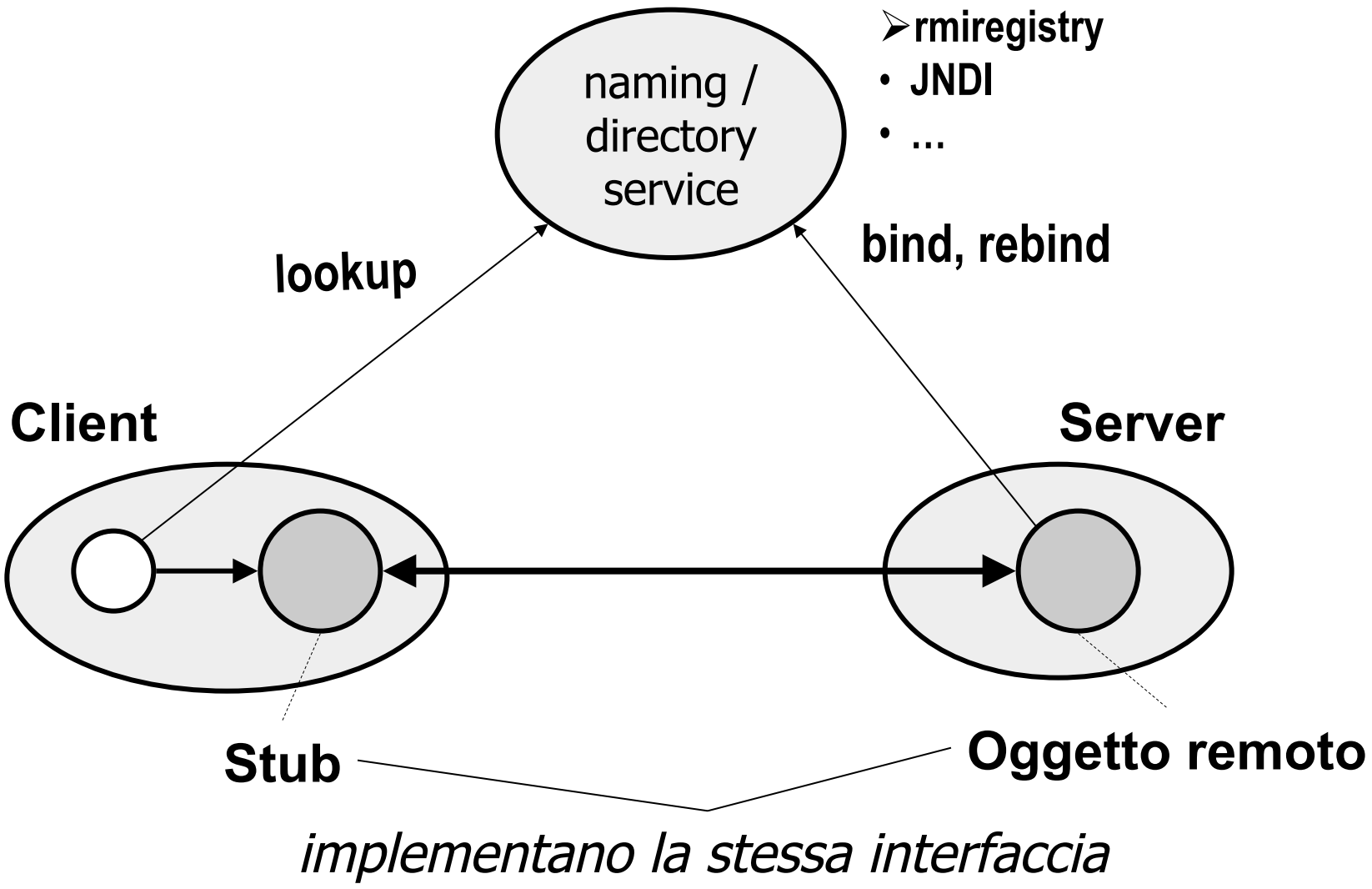
Il client

1. **interroga** l'**rmiregistry** usando il nome pubblico ed ottiene uno **stub** dell'oggetto remoto
2. **invoca** i metodi dello stub

Il server

1. **crea** ed **esporta** un oggetto e lo **registra** sotto un nome pubblico per mezzo di rmiregistry
2. attende le invocazioni di metodo, le esegue e così via all'infinito

Una visione d'insieme



Realizzazione di un oggetto remoto

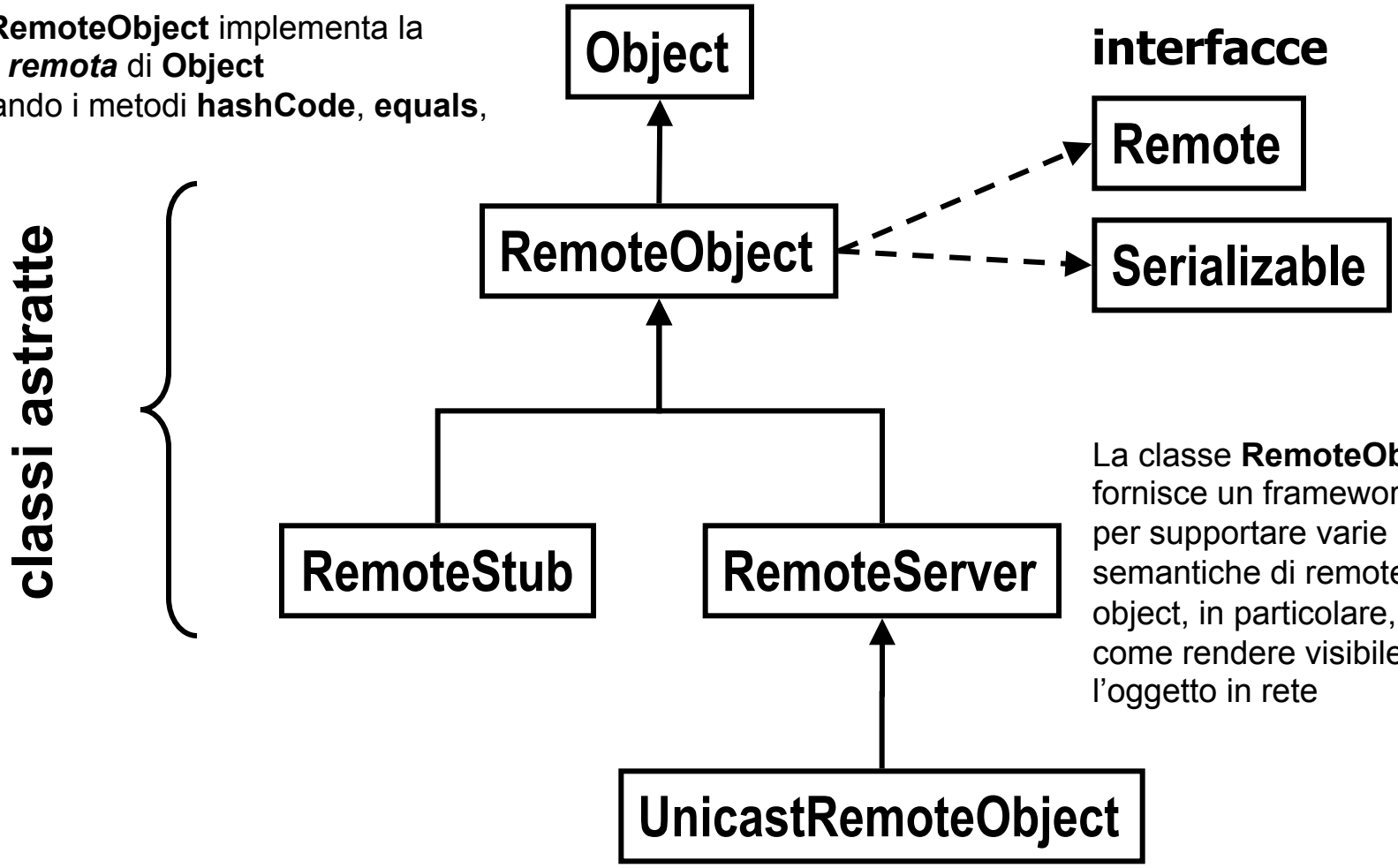
- **Le classi**
  - **Calculator.java** (interfaccia remota)
  - **CalculatorImpl.java** (oggetto remoto)
  - **CalculatorServer.java** (server)
  - **CalculatorClient.java** (client)
- **Compilazione**
  - **javac Calculator*.java**
- **Generazione stub (& skeleton)**
  - **rmic CalculatorImpl.class** (genera automaticamente lo stub **CalculatorImpl_Stub.class**)
  - *A partire da Java SE 5 questo passo non è più necessario*

Realizzazione di un oggetto remoto

- **Esecuzione (Windows)**
  - **start rmiregistry**
    - *Eseguire questo comando nella directory che contiene l'interfaccia e lo stub* (**Calculator.class**, **CalculatorImpl_Stub.class**)
  - **start java CalculatorServer**
  - **start java CalculatorClient**
    - La classe **Calculator.class** deve essere nel classpath del cliente a meno che non si usi il caricamento remoto delle classi

La gerarchia delle classi

La classe **RemoteObject** implementa la ***semantica remota*** di **Object** implementando i metodi **hashCode**, **equals**, e **toString**



La classe **RemoteObject** fornisce un framework per supportare varie semantiche di remote object, in particolare, come rendere visibile l'oggetto in rete

La classe UnicastRemoteObject

- La classe **UnicastRemoteObject** permette di definire oggetti **remoti, non replicati**, **volatili**, **non rilocabili** e basati sui flussi TCP
  - ***non replicato***: esiste una sola copia dell'oggetto
  - ***volatile***: il riferimento remoto all'oggetto è valido solo mentre il processo server è attivo
  - ***non rilocabile***: l'oggetto remoto non può migrare da un server all'altro
- Tipicamente, gli oggetti remoti estendono **RemoteObject**, attraverso **UnicastRemoteObject**; altrimenti, la classe implementazione deve implementare **hashCode**, **equals**, e **toString** in modo appropriato per gli oggetti remoti

Esportazione di un oggetto remoto

- ***Un oggetto remoto deve essere esportato*** in modo da accettare richieste ascoltando invocazioni remote di metodo da parte dei clienti su di una porta (anonima)
- **Esportazione implicita**. Se la classe remota estende **UnicastRemoteObject**, un oggetto viene implicitamente esportato al momento della creazione
  - Il costruttore di **UnicastRemoteObject** esegue l'esportazione
  - Il costruttore dichiara **RemoteException** nella sua clausola **throws**
- **Esportazione esplicita**. Se la classe remota non estende **UnicastRemoteObject**, l'oggetto remoto deve essere esportato esplicitamente invocando **UnicastRemoteObject.exportObject**

Esportazione esplicita

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

public class Calculatormpl  implements Calculator {

        public CalculatorImpl(String n) throws RemoteException {
            name = n;
            UnicastRemoteObject.exportObject(this);
    }
        // il resto
}
```

La classe UnicastRemoteObject

- **protected UnicastRemoteObject() throws RemoteException** crea ed esporta questo oggetto usando una porta anonima
- **protected UnicastRemoteObject(int port) throws RemoteException** crea ed esporta questo oggetto usando la porta specificata
- **static RemoteStub exportObject(Remote obj) throws RemoteException** esporta l'oggetto remoto **obj** e lo abilita a ricevere richieste usando una porta anonima
- **static RemoteStub exportObject(Remote obj, int port) throws RemoteException** esporta l'oggetto remoto **obj** e lo abilita a ricevere richieste usando la porta **port**

Parametri e valore di ritorno

Trasmissione dei parametri e del valore di ritorno

- **tipo primitivo**: la trasmissione per valore in un formato esterno indipendente dalla piattaforma (*trasmissione per valore*)
- **riferimento ad oggetto**: viene trasmesso l'oggetto in forma serializzata (*trasmissione per valore*)
- **riferimento ad oggetto remoto**: viene trasmesso lo stub (*trasmissione per riferimento*)

Esempio: Hello

Serializzazione in RMI

Annotazione della classe

- Quando un oggetto viene trasmesso in una RMI, la serializzazione viene estesa con il ***meccanismo di annotazione della classe***

  Annotazione della classe: nel flusso, le informazioni relative alla classe sono estese per mezzo di informazioni aggiuntive (URL) che indicano da dove la classe (**.class**) può essere caricata

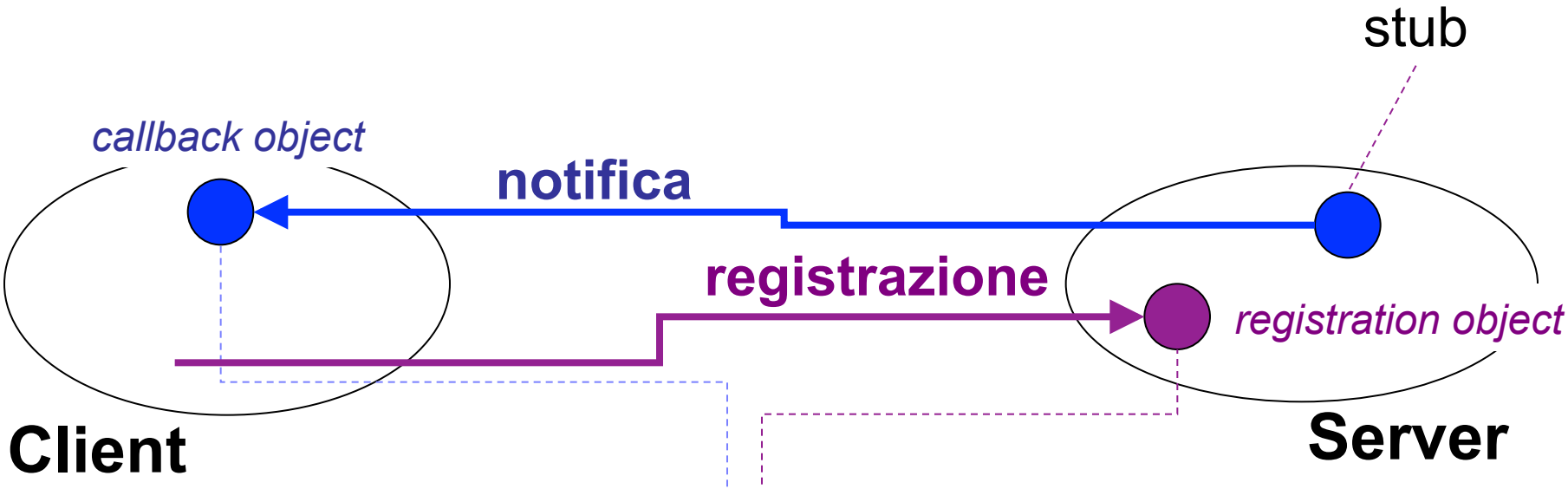
Callback

- ***Problema. Il cliente si deve sincronizzare con eventi che si verificano sul server***
    - **Esempio**. Ad esempio, un server offre un'interfaccia di rete ad un servizio di back-end. Questo servizio richiede un tempo lungo di esecuzione. Il server monitorizza l'esecuzione del servizio. Il cliente vuole essere avvisato quando si verificano gli eventi e_1= 25% del lavoro svolto, e_2=50% del lavoro svolto (analogamente si può definire sul tempo).
- ***Possibili approcci***
    - ***Polling***
    - ***Callback***

Polling e callback

- ***Polling***
  - ***Idea***. Il client chiede periodicamente al server se l'evento di interesse si è verificato
  - ***Vantaggi***.
    - Semplice da realizzare
  - ***Svantaggi***.
    - **Prestazioni**. Aumenta il traffico di rete ed il carico sul server
    - **Tempestività**. Si può ridurre la frequenza di polling ma si perde in tempestività
- ***Callback***.
  - ***Idea***. Quando l'evento si verifica, il server informa il client.
  - ***Svantaggi.*** Realizzazione più complessa.
  - ***Vantaggi***. Il callback risolve gli svantaggi del polling

Callback con RMI



- Il client implementa un ***callback object*** (oggetto remoto) che il server utilizza per **notificare** eventi

- Il server fornisce un ***registration object*** (oggetto remoto), tramite il quale un client può **registrare** un **callback object**
  - *Quando si verifica l'evento, il server invoca il metodo remoto di notifica dell'evento*

Callback con RMI

```
// Interfaccia di registrazione del registration object
import java.rmi.*;

public interface Registration extends Remote
{
    public void register(Callback em)  throws RemoteException
    public void unregister(Callback em) throws RemoteException
}

// Interfaccia del callback object
import java.rmi.*;

public interface Callback extends Remote
{
    public void notify(Event e) throws RemoteException
}
```

Problemi con la callback

Il server necessita di una lista aggiornata dei clienti registrati

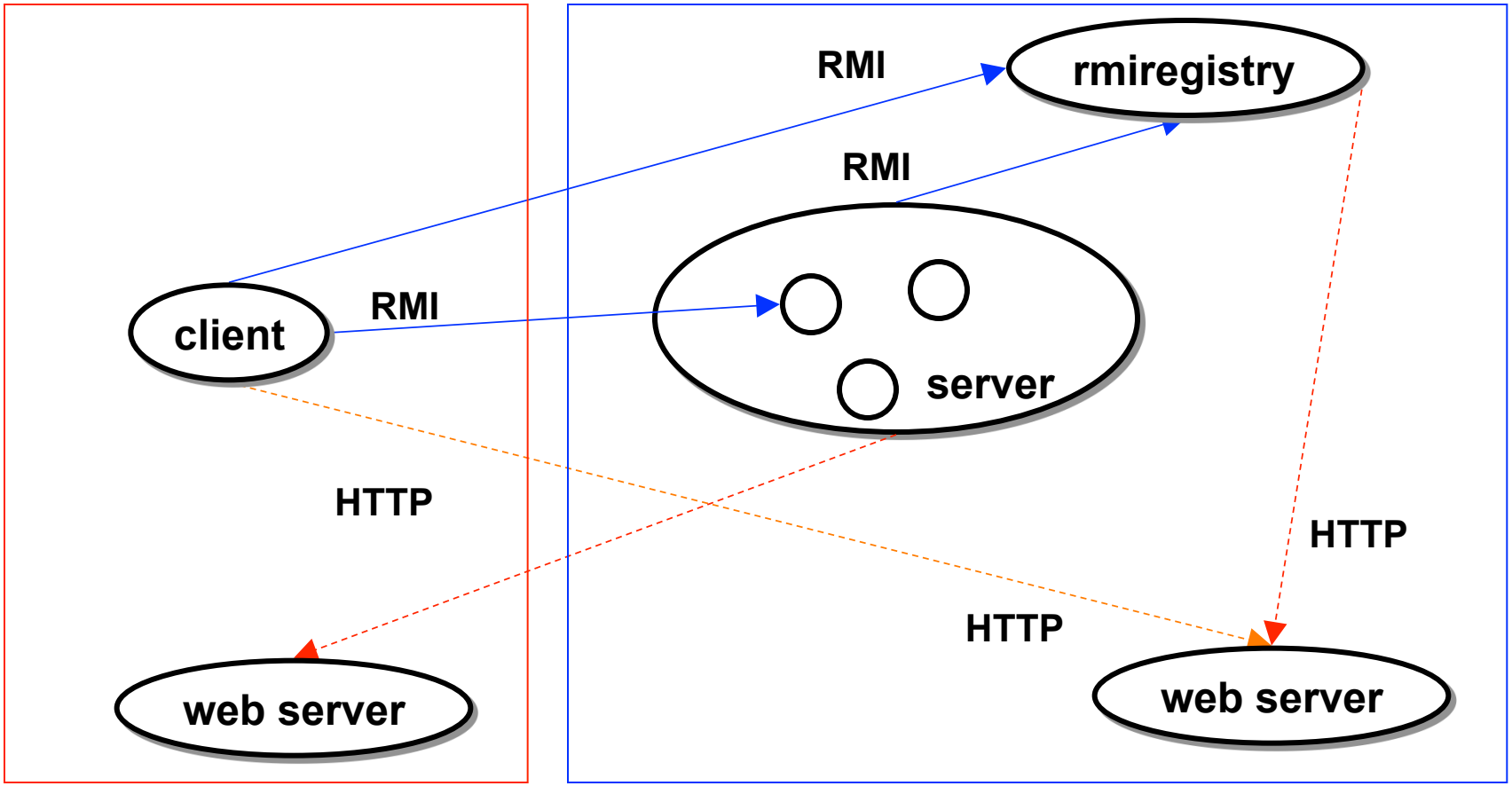
- Meccanismo di *lease* permette di gestire i casi in cui i clienti terminano senza notificare il server

Il server esegue una serie di invocazioni sincrone di RMI verso i callback object dei clienti

- La notifica ad un cliente è ritardata perché la connessione verso un client è guasto oppure il cliente è guasto o disattivato
- Il client ed il server sono disaccoppiati da una mailbox

Il caricamento remoto delle classi

Una visione d'insieme



Attributi di un programma Java

- Un programma Java viene eseguito in un **ambiente di esecuzione** caratterizzato da **attributi**
  - **Attributo = (nome, valore)**
  - **Esempio: (os.name, Solaris)**
- ***Tipi di attributo***
  - **Attributi di sistema** (macchina host, utente, directory corrente, …), gestiti dalla classe **System**
  - **Attributi di programma** (preferenze) configurabili (opzioni di avvio, dimensione delle finestre,…)

Attributi di un programma

- **Un programma può impostare gli attributi di programma mediante**:
- **Proprietà**, che definiscono attributi che persistono (stream) tra invocazioni successive di un programma (classe **java.util.Properties**)
- **Argomenti della linea di comando**, che definiscono attributi in modo non persistente, cioè che valgono per una singola esecuzione di un programma
- **Parametri delle applet**, che sono simili agli argomenti della linea di comando ma sono utilizzati con le applet e non con le applicazioni

Proprietà e classe Properties

- Java utilizza le ***proprietà*** per gestire gli attributi
  - Le proprietà sono realizzate dalla classe **java.util.Properties** che gestisce coppie *nome-valore*
  - ***Attributi di sistema****:* sono gestiti dalla classe **System** per mezzo della classe **Properties**
  - ***Attributi di programma***: un programma Java può utilizzare direttamente la classe **Properties** per gestire gli attributi di programma
  - L'accesso alle proprietà è soggetto all'approvazione da parte del ***Security Manager***

Codebase

- Il caricamento remoto delle classi in RMI è controllato dalle seguenti proprietà:
- **proprietà java.rmi.server.codebase**
  - Specifica l'**URL** della locazione che fornisce le classi per gli oggetti che sono *inviati* da questa JVM
    - L'URL può specificare vari protocolli per scaricare le classi: **file:**, **ftp:**, **http:,…**
  - Quando serializza un oggetto, RMI inserisce l'URL nel flusso (***annotazione della classe***)
  - La JVM che riceve l'oggetto serializzato, se ha bisogno di caricare la relativa classe, può caricarla dalla locazione specificata dall'URL
  - **NOTA BENE**: ***JVM non invia la classe con l'oggetto serializzato***

continua…

Codebase

- Il ***codebase*** specifica la locazione da cui si caricano le classi nella JVM
  - Il **classpath** è il codebase locale
  - La proprietá **java.rmi.server.codebase** permette di specificare un codebase remoto

Security Manager

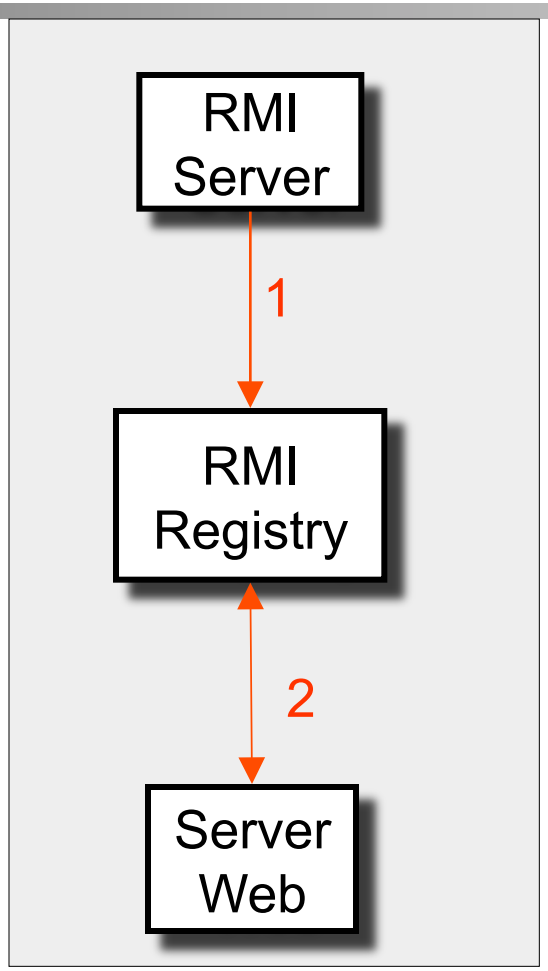
- **SM** è responsabile di controllare a quali risorse una JVM ha accesso
  - Per default, un'applicazione standalone non ha SM mentre un applet usa il SM del browser in cui gira
- SM determina se il codice caricato remotamente può accedere al file system oppure eseguire operazioni privilegiate
  - **Se SM non è definito, il caricamento remoto non è permesso**
  - Una JVM che deve caricare classi remotamente deve
    - istanziare un SM
    - definire una *politica di sicurezza* (insieme di permessi)
- **RMISecurityManager** è il SM di default per RMI e garantisce gli stessi controlli del SM di un browser

Proprietá in Java RMI (semplificato)

continua

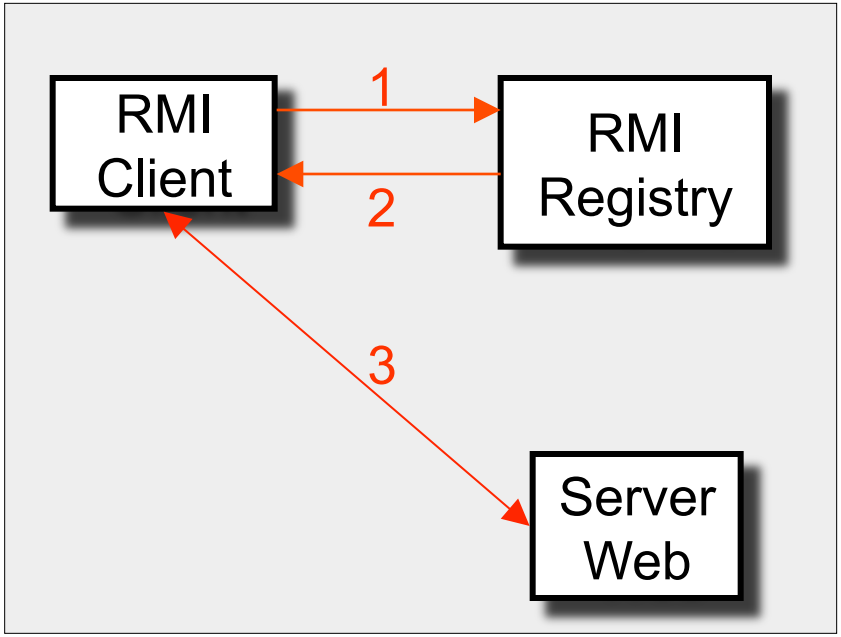
- **proprietà java.security.policy**
  - Il codice, locale o remoto, è soggetto ad una ***politica di sicurezza***
  - La politica di sicurezza definisce un insieme di permessi concessi ai vari moduli di codice per accedere alle risorse del sistema
  - La proprietà **java.security.policy** specifica il *file delle politiche* che estende la politica di default per il controllo degli accessi

Caricamento remoto delle classi



1. RMI Server segue la **[re]bind,** specificando il **codebase** impostato tramite la proprietà **java.rmi.server.codebase**
2. RMI Registry ***carica*** la classe dello stub dalla locazione specificata dal codebase

Caricamento remoto delle classi



1. RMI Client invoca **Naming.lookup**
2. RMI Registry ritorna un'**istanza** dello stub
3. RMI Client ***preleva*** la classe dello stub (**_stub.class**) dall'**URL** specificato dal **codebase**

Esempio Compute

- Realizzazione di un ***Compute Service*** che riceve un task, lo esegue e ne ritorna il risultato
- **Compute** object (remoto) ha un'unica operazione

  Object executeTask(Task t)

- Un **Task** ha un'unica operazione

  Object execute()

Esecuzione: attivazione di rmiregistry

unset CLASSPATH

start rmiregistry

RMI Registry deve essere attivato in una shell che non ha variabile **CLASSPATH** oppure in cui tale variabile non specifica il path di alcuna classe, inclusa quella che si vuole caricare remotamente

Altrimenti, RMI Registry "si dimenticherà" che le classi devono essere caricate dal codebase specificato da **java.rmi.server.codebase** e, perciò, non invierà al cliente il vero codebase e quindi il cliente non sarà capace di localizzare e caricare le classi

Esecuzione: attivazione di client e server

Attivazione di RMI Server

```
java –Djava.rmi.server.codebase=http://dini.iet.unipi.it:2002/
      -Djava.security.policy=rmi/compute/engine/java.policy
      ComputeEngine
```

Attivazione di RMI client

```
java -Djava.rmi.server.codebase=http://dini.iet.unipi.it:2002/
      -Djava.security.policy=rmi/compute/client/java.policy
       ComputePi dini.iet.unipi.it 20
```

ClassFileServer

Il **ClassFileServer** è un HTTP Server semplificato che fornisce il servizio di caricamento delle classi

porta su cui il server ascolta

java ClassFileServer 2002 D:\home\didattica\tiga\materiale-didattico\java\rmi\classfileserver

classpath

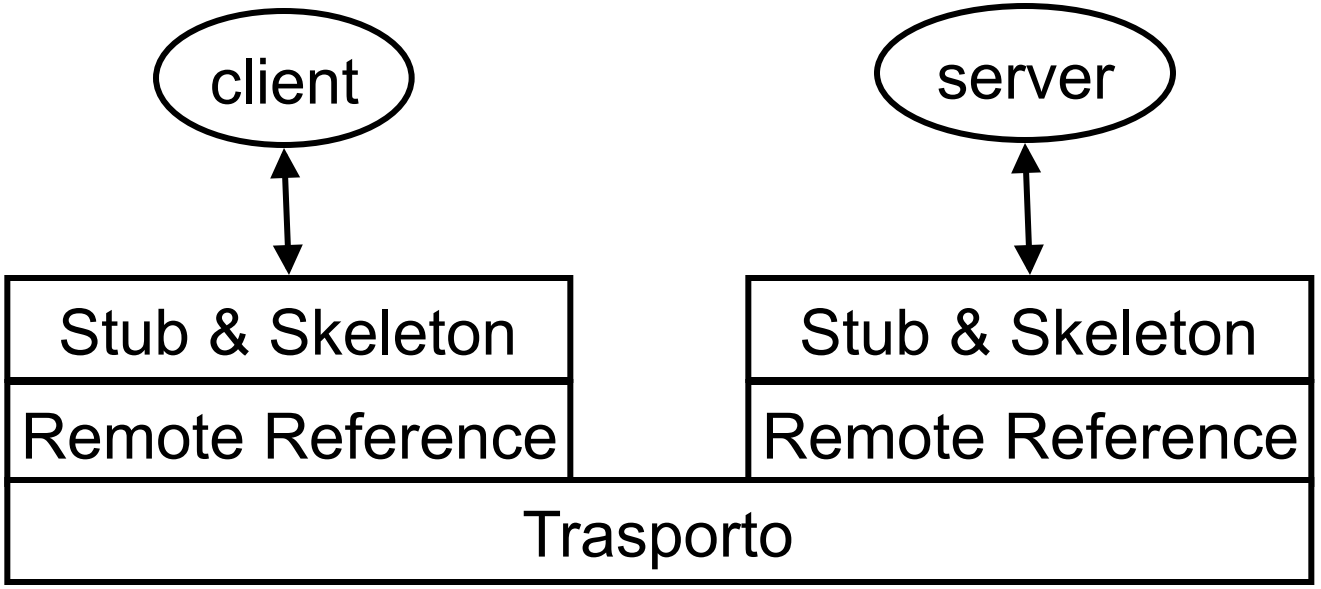
RMI e threads

- RMI non specifica se l'esecuzione di un metodo remoto avviene in un thread separato oppure no
- Siccome l'esecuzione di un metodo remoto **può** avvenire in un thread separato, l'implementazione dell'oggetto remoto deve essere **thread-safe**

RMI

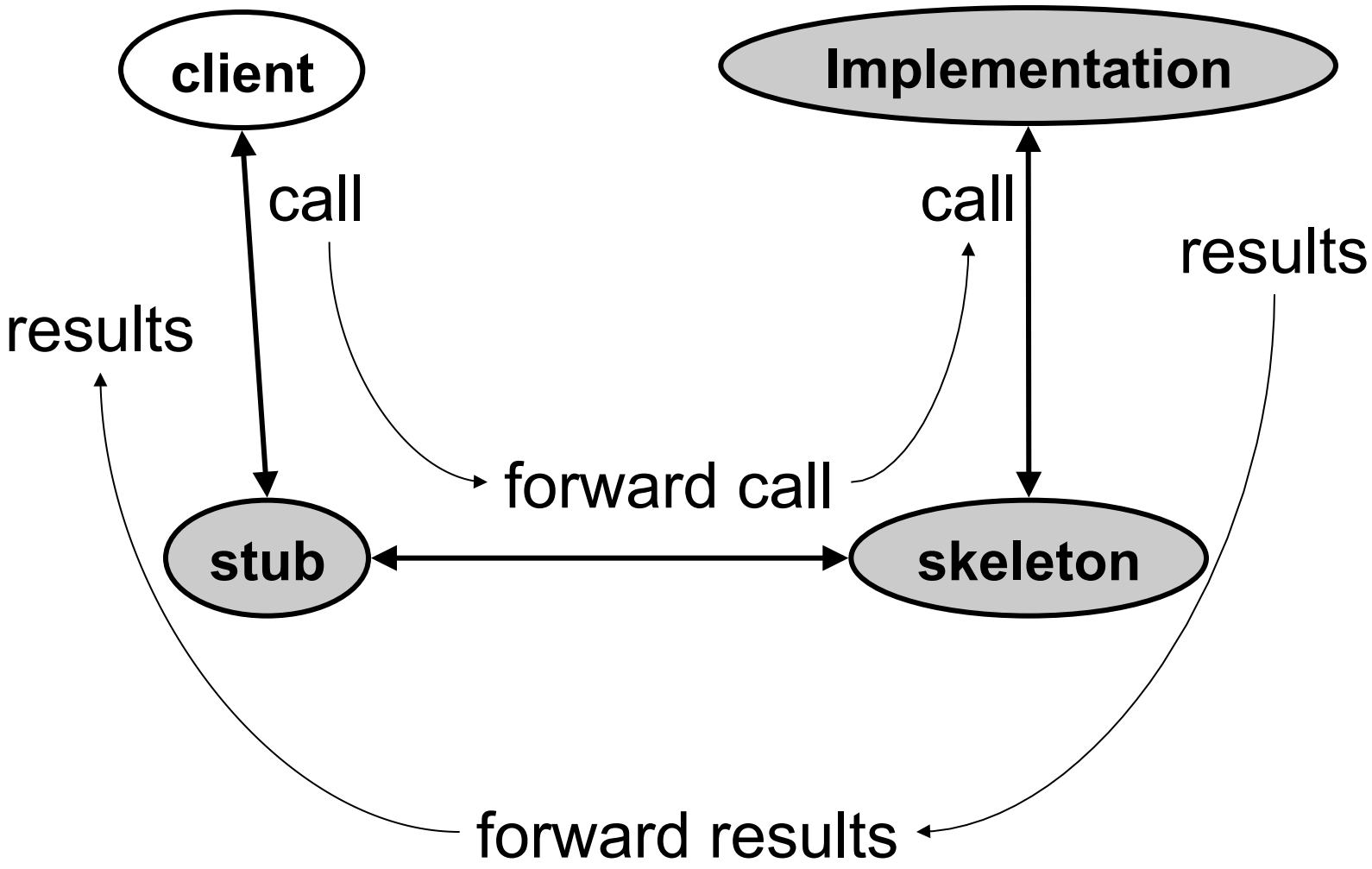
CENNI IMPLEMENTATIVI

Architettura di RMI



Ciascun livello può essere sostituito o esteso senza modificare gli altri

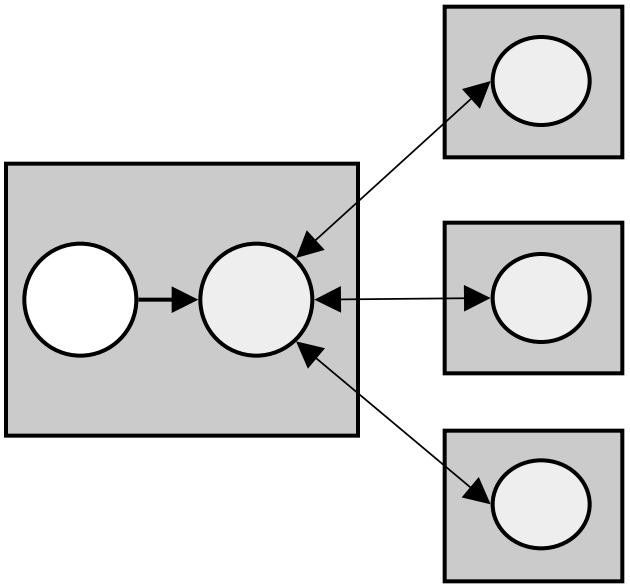
Il livello Stub & Skeleton



Il Livello Remote Reference

Il livello **Remote Reference** implementa la semantica di Java RMI

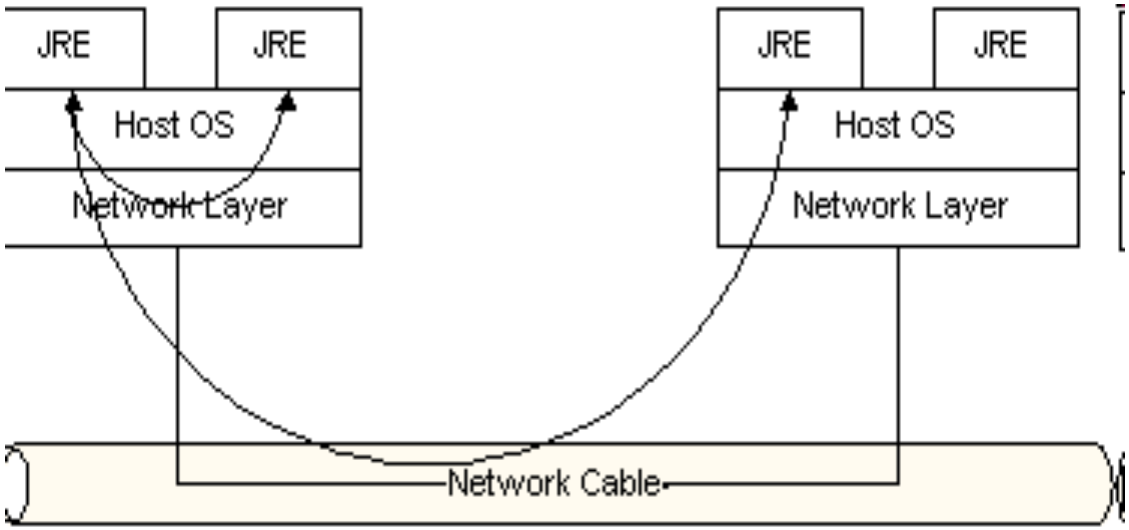
- Oggetti unicast, volatili, at-most-once (JDK 1.1)
- Oggetti attivabili (JDK 2)
- Altre semantiche possibili: multicast



Il proxy

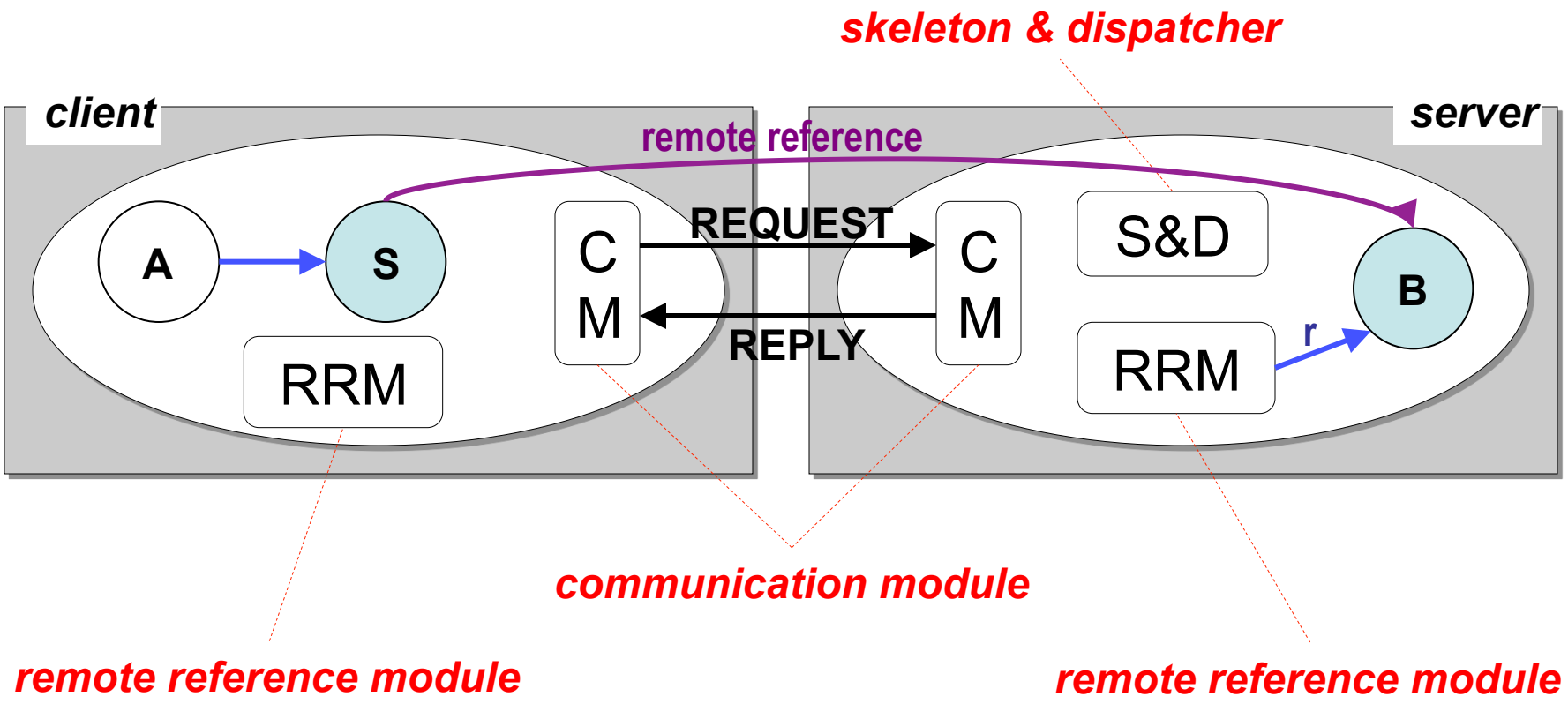
- inoltra simultaneamente la richiesta ad un insieme di implementazioni ed
- accetta la prima risposta

Il livello di trasporto



- Tutte le connessioni sono TCP/IP
- Java RMI implementa Java Remote Method Protocol (**JRMP**) su TCP

Implementazione di RMI



RMI processing

- Siano:
  - **B** un oggetto remoto
  - **m** un metodo della sua interfaccia remota
  - **R** il riferimento remoto a **B**
  - **S** uno stub che incapsula **R**
  - **r** il riferimento locale al server dell'oggetto remoto **B**
- RMI processing deve affrontare due aspetti:
  - l'invocazione (attraverso **S**) del metodo remoto **m** di **B**
  - l'oggetto remoto **B** viene trasmesso come argomento o valore di ritorno

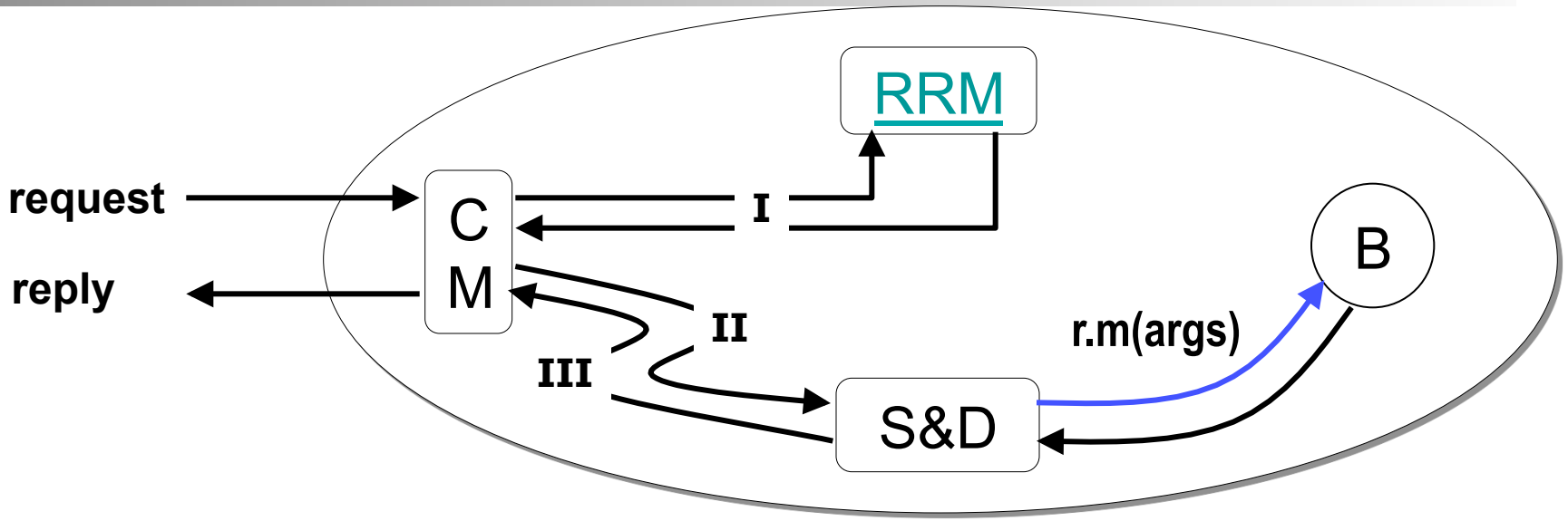
Stub & Communication Module

- ***Proxy/stub***
  - Il proxy/stub rende la RMI trasparente all'oggetto cliente
  - Il proxy/stub ha la stessa interfaccia dell'oggetto remoto ed inoltra a questi l'invocazione di metodo
  - Il proxy/stub nasconde il riferimento remoto ed esegue marshalling ed unmarshalling (lato client)
  - C'è un proxy/stub per ogni oggetto remoto di cui il processo ha un riferimento remoto
- ***Communication Module (CM)***
  - esegue il protocollo request-reply e realizza la semantica (at-most-once, at-least-once, …) della RMI

Skeleton & Dispatcher (S&D)

- Un modulo S&D per ciascuna classe remota
- Il modulo S&D ha il compito di:
  - determinare il metodo da invocare;
  - eseguire l'unmarshalling degli argomenti;
  - eseguire l'invocazione locale del metodo;
  - eseguire il marshalling dei risultati;
  - preparare il messaggio di reply
- Il modulo S&D è generato dinamicamente a partire dall'interfaccia remota

Request processing



I. Alla ricezione del *request message*, CM invia il riferimento remoto **R** ad RRM, il quale reperisce in *Remote Object Table* (*ROT*) il corrispondente riferimento locale **r** all'oggetto B e lo ritorna a CM

II. CM determina il modulo S&D sulla base del valore del campo Remote Class/Interface del riferimento remoto **R** e passa a tale modulo la coppia **(request message, r)** (*continua*)

Request processing: S&D

III. Alla ricezione della coppia **(request message, r)**, il modulo S&D esegue le seguenti azioni:

   a. determina il metodo **m** da invocare sulla base del valore del campo **Method Identifier** del **Request Message**

   b. esegue l'unmarshaling degli argomenti **args** contenuti nel campo **Arguments** del **Request Message**

   c. invoca il metodo **results = r.m(args)**

   d. esegue il marshalling dei valori di ritorno **results**

   e. prepara il **Reply Message** e lo passa a CM

- **N.B.: l'elaborazione di una chiamata remota assume che il mapping R ←→ r sia già disponibile nella tabella ROT di RRM**

Remote Reference Module (RRM)

- RRM crea i riferimenti remoti
- RRM traduce **rref ➔lref**
- RRM mantiene la ***Remote Object Table*** (***ROT***)

Remote Object Table

riferimento remoto	riferimento locale
R	r

riferimento *locale* a:
- oggetto remoto (server)
- stub (client)

Trasmissione dei parametri

- ***L'oggetto viene trasmesso come argomento (request message)***
  - Lo stub trasmette al modulo RRM il riferimento locale **r** all'oggetto il quale ritorna il riferimento remoto **R** da inserire nel **request** message;
- ***L'oggetto viene trasmesso come valore di ritorno (reply message)***
  - Il S&D trasmette al modulo RRM il riferimento locale **r** all'oggetto il quale ritorna il riferimento remoto **R** da inserire nel messaggio;

Il binder

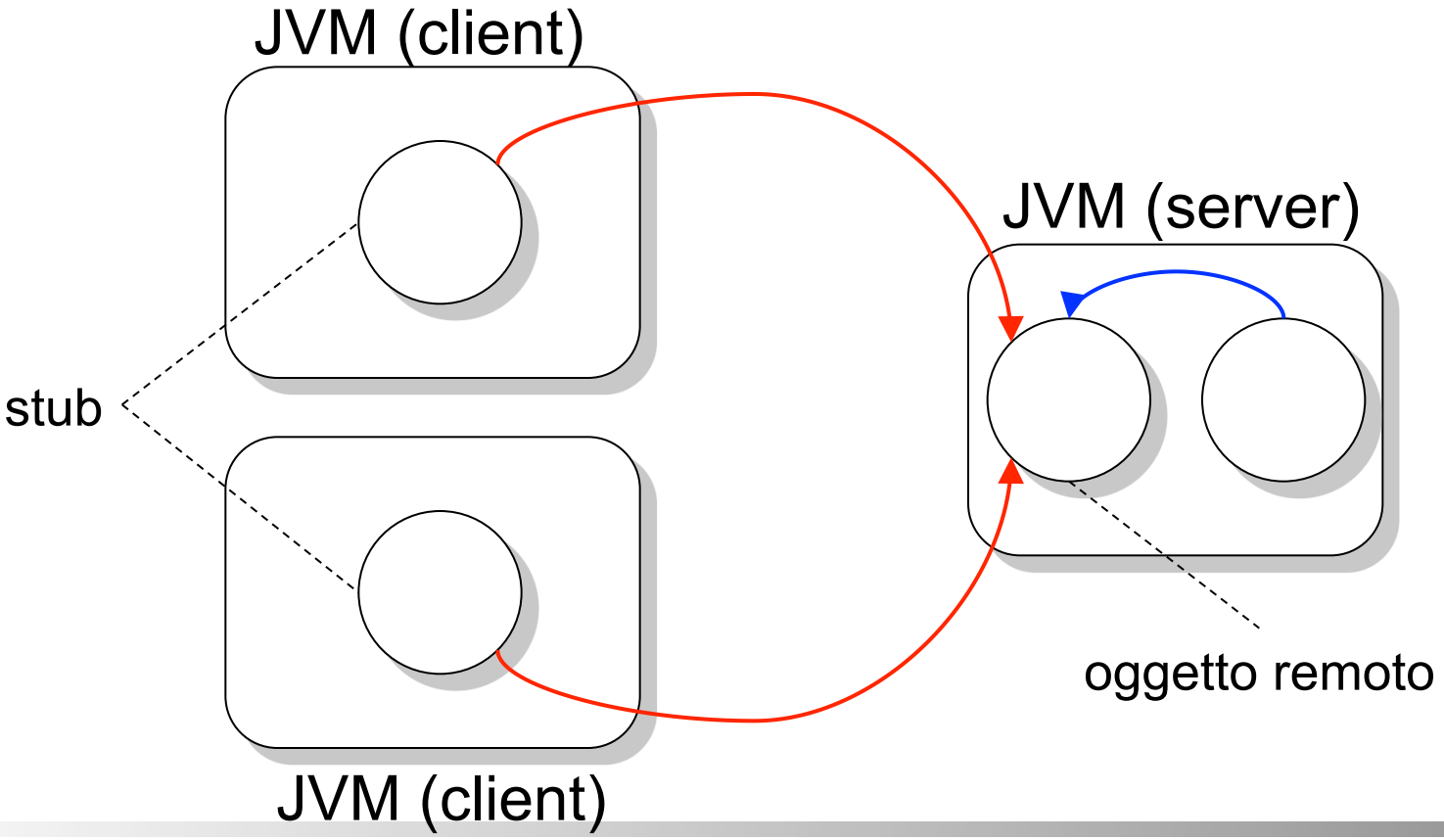
- il binder è un servizio (directory service) separato che mantiene la corrispondenza tra nomi (stringhe) e riferimenti remoti
- un server registra un proprio oggetto remoto sotto un certo nome
- I client ottengono i riferimenti remoti a tale oggetto facendo una ricerca per nome

RMI

DISTRIBUTED GARBAGE COLLECTION

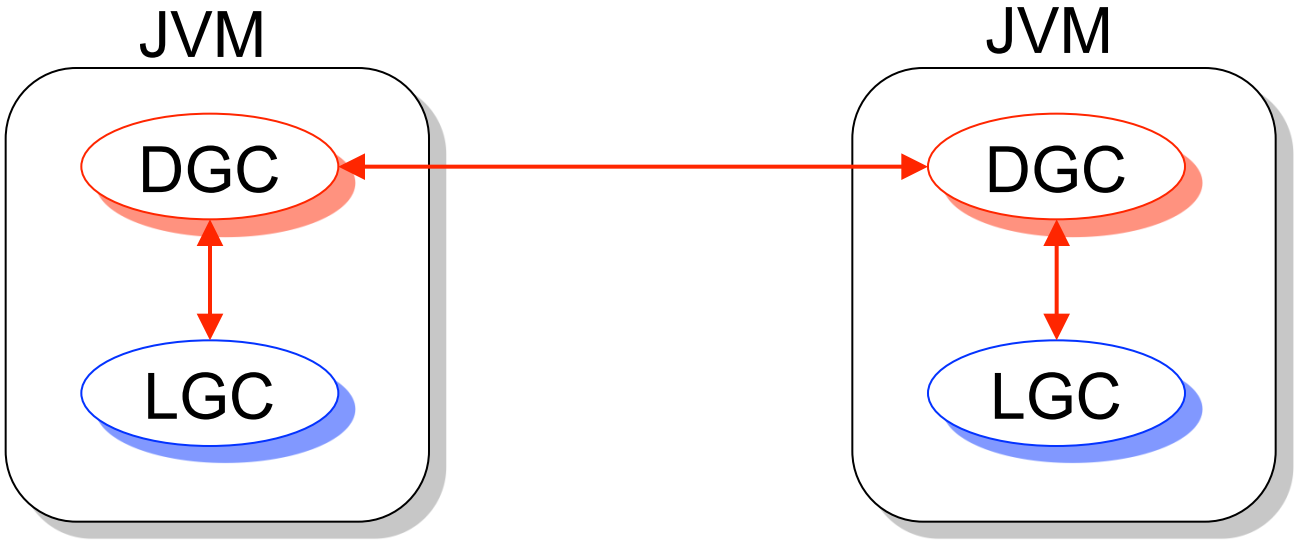
Garbage Collection Distribuito

- Un oggetto può essere raccolto quando non ci sono più riferimenti, **locali** o **remoti**, ad esso



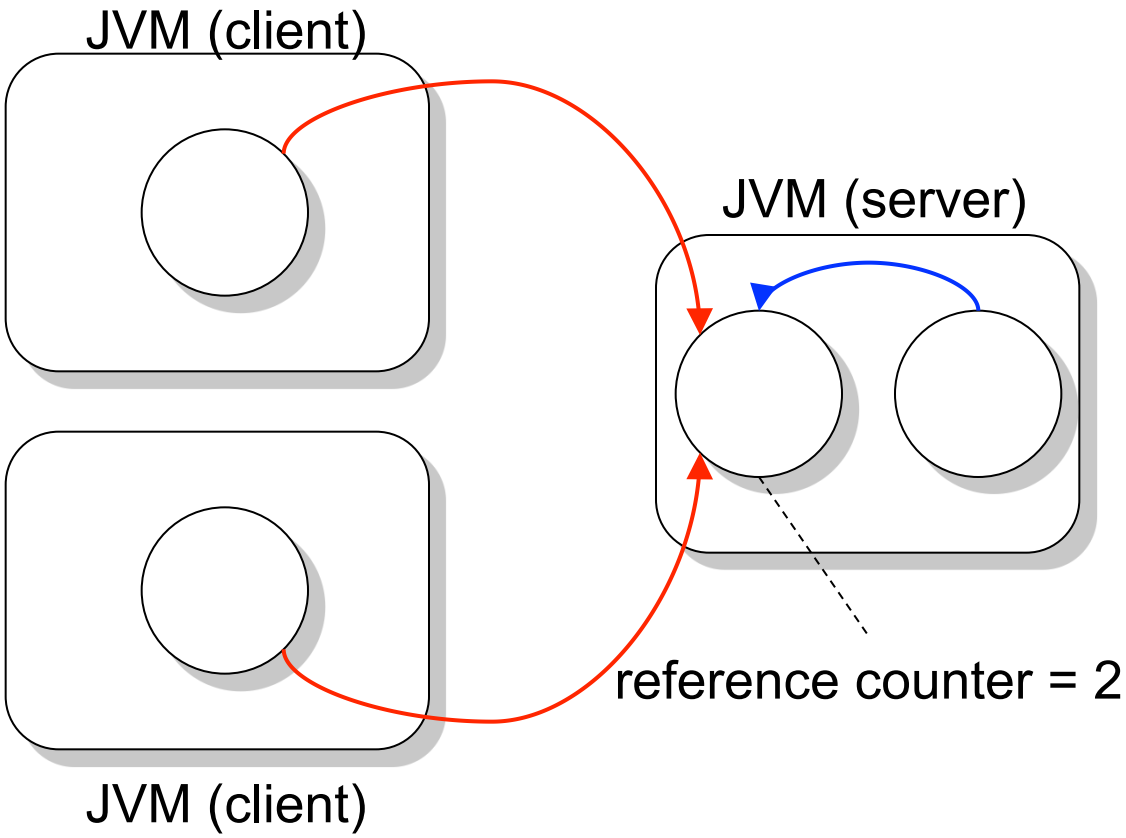
Garbage Collection Distribuito

- In Java, il servizio **Distributed Garbage Collector** (**DGC**) collabora con il servizio Local Garbage Collector (LGC)



Garbage Collection Distribuito

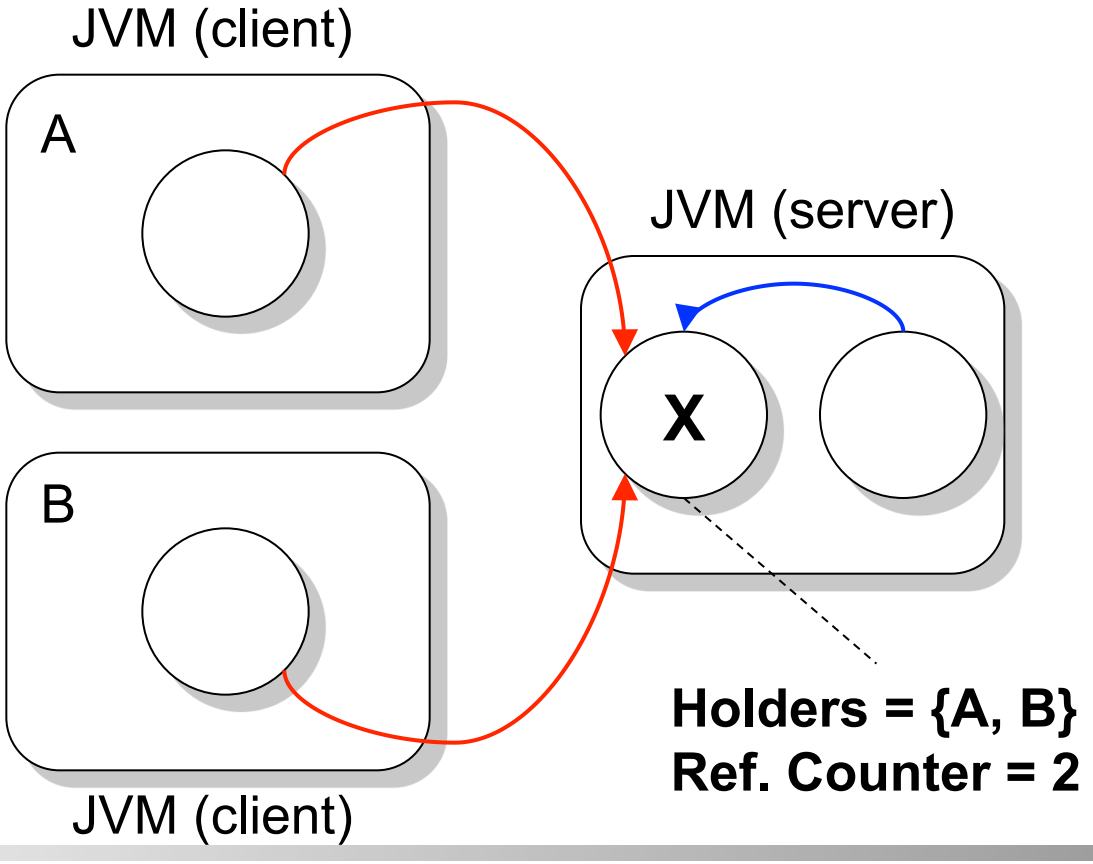
- DGC è basato sul meccanismo del **reference counting**, cioè DGC tiene traccia dei riferimenti a ciascun oggetto remoto



- DGC viene informato di ogni creazione e distruzione di stub in modo da poter incrementare e decrementare, rispettivamente, il reference counter

Garbage Collection Distribuito

- Ogni oggetto remoto X è associato all'insieme **holders** che memorizza l'insieme dei clienti che detengono un riferimento all'oggetto stesso



Garbage Collection Distribuito

ALGORITMO (semplificato)

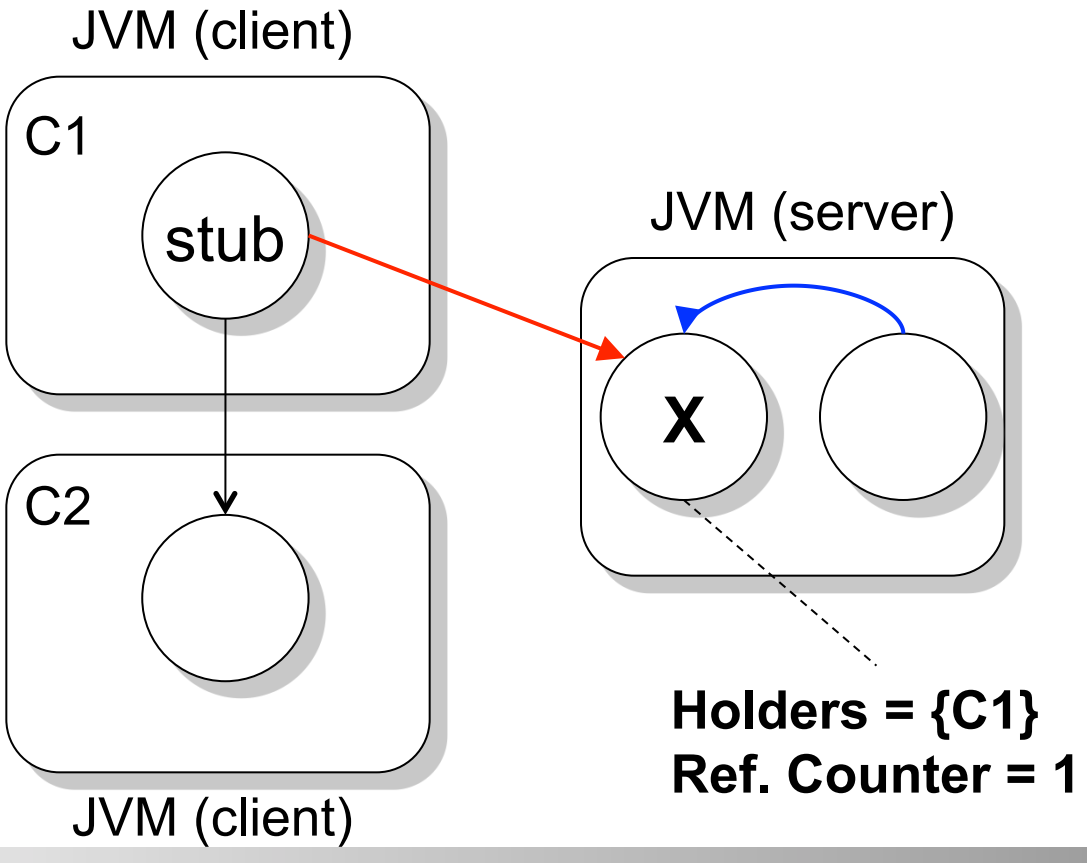
- Quando riceve un riferimento remoto all'oggetto **X**, il cliente C invoca l'operazione remota **addRef**(**C**) sul server e poi crea lo stub **S(X)**; il server aggiunge *C* a **X.holder**

- Quando il LGC di C, LGC_C, rileva che lo stub **S(X)** può essere raccolto, LGC_C invoca l'operazione **removeRef**(**C**) e cancella lo stub; il server toglie *C* da **X.holder**

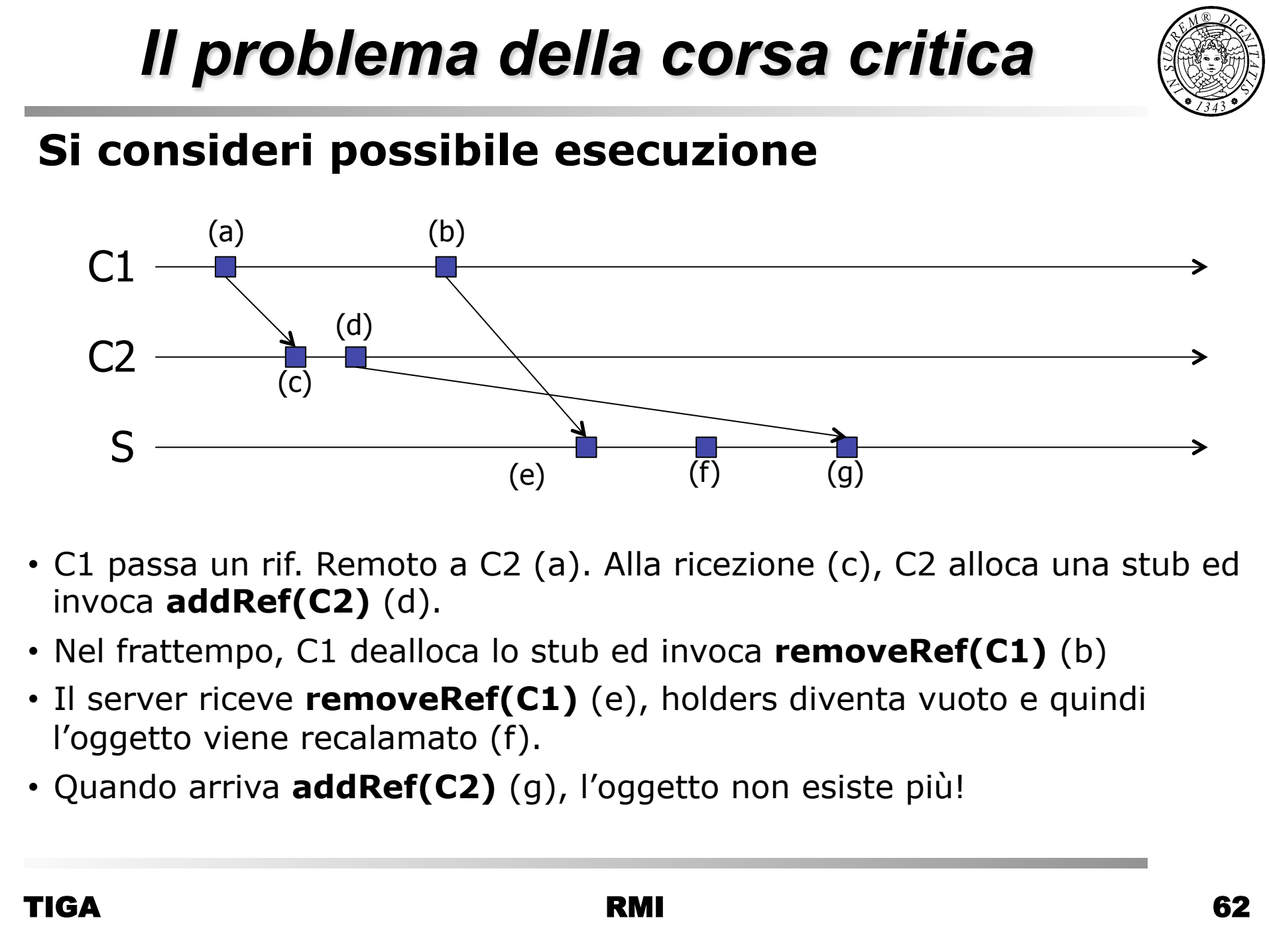
- Quando **X.holder** è vuoto (**ref. Counter = 0**) e non ci sono riferimenti locali a **X**, l'oggetto **X** può essere reclamato

Il problema della corsa critica

- Si consideri questa situazione: C1 passa a C2 un riferimento remoto ad X come parametro o valore di ritorno



Il problema della corsa critica

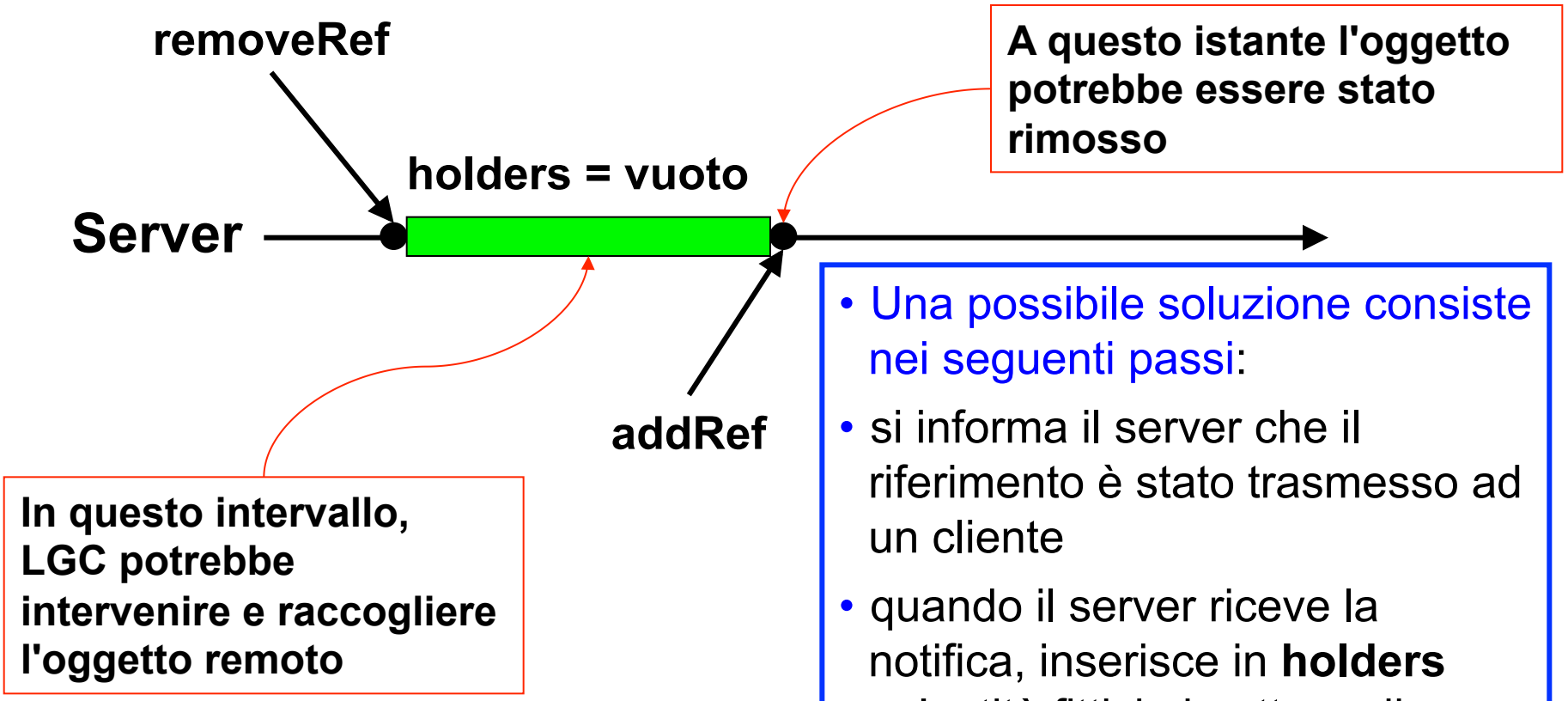


Si consideri possibile esecuzione

- C1 passa un rif. Remoto a C2 (a). Alla ricezione (c), C2 alloca una stub ed invoca **addRef(C2)** (d).
- Nel frattempo, C1 dealloca lo stub ed invoca **removeRef(C1)** (b)
- Il server riceve **removeRef(C1)** (e), holders diventa vuoto e quindi l'oggetto viene recalamato (f).
- Quando arriva **addRef(C2)** (g), l'oggetto non esiste più!

Garbage Collection Distribuito

- IL PROBLEMA DELLA CORSA CRITICA



- Una possibile soluzione consiste nei seguenti passi:
- si informa il server che il riferimento è stato trasmesso ad un cliente
- quando il server riceve la notifica, inserisce in **holders** un'entità fittizia in attesa di **addRef**

Garbage Collection Distribuito

- Le azioni dell'algoritmo sono "nascoste" nella implementazione dell'oggetto remoto
- Un'applicazione server può ricevere una notifica dell'evento **holders = vuoto** implementando l'interfaccia **java.rmi.server.Unreferenced**
  - Questa interfaccia ha un solo metodo, **void unreferenced()**, che viene invocato dal RMI runtime

Garbage Collection Distribuito

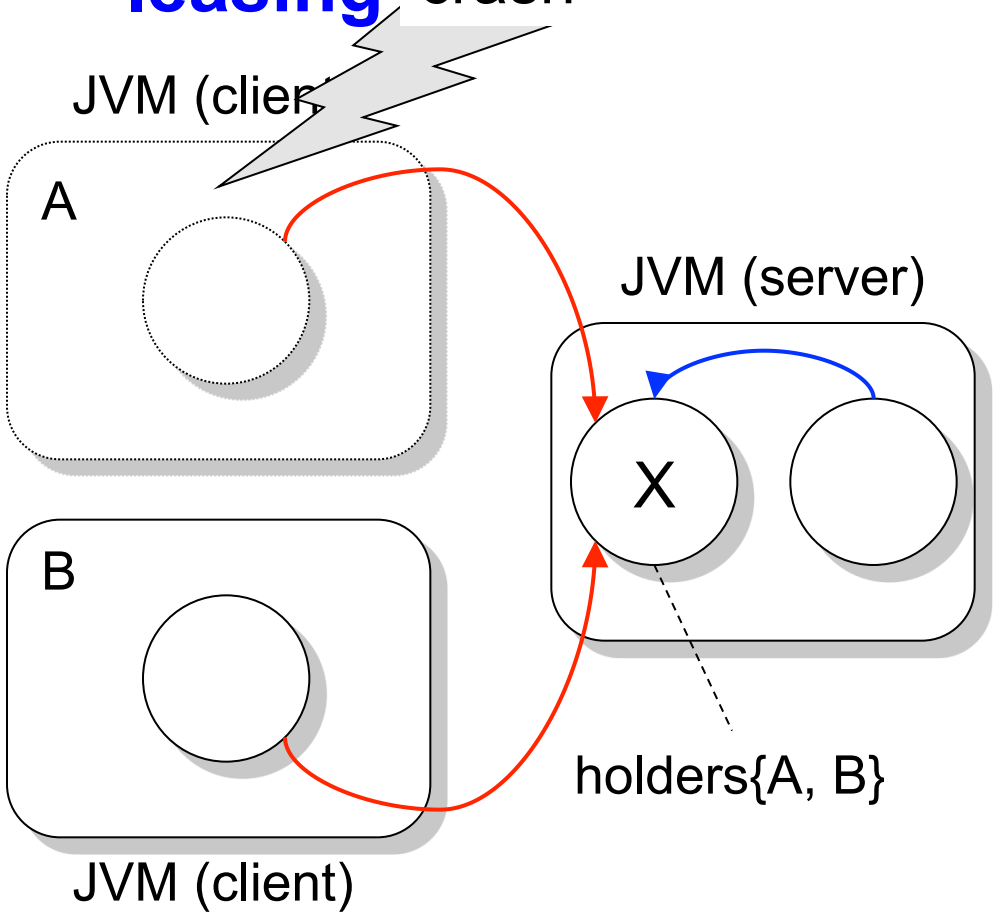
- Le operazioni **addRef** e **removeRef**
- sono chiamate di procedura remota che seguono la semantica **at-most-once**
- non richiedono una sincronizzazione globale
- non interagiscono con le RMI del livello applicativo perché sono invocate solo quando viene creato o distrutto uno stub

Garbage Collection Distribuito

- **Il servizio DGC di Java tollera i guasti di comunicazione**
- Le operazioni **addRef** e **removeRef** sono idempotenti
- Se l'invocazione di **addRef** ritorna un'eccezione, il cliente non crea lo stub ed invoca l'operazione **removeRef**
- L'operazione **removeRef** ha sempre successo
- Se l'invocazione di **removeRef** dà luogo ad una eccezione interviene il meccanismo di **leasing**

Garbage Collection Distribuito

Il meccanismo del leasing



IL PROBLEMA

- Se il client va in crash non esegue **removeRef**, perciò **Holders** non diventa mai vuoto e perció l'oggetto **X** non sarà mai reclamato

Il LEASING

- Un riferimento remoto non viene dato per sempre ma ***affittato*** per un periodo di tempo massimo prefissato
- Quando il periodo è trascorso, il riferimento viene rimosso
- L'affitto è rinnovabile

Garbage Collection Distribuito

- Il meccanismo di leasing
  - Il server dà un riferimento remoto in affitto (leasing) a ciascun cliente per un tempo massimo prefissato **T**
  - Il periodo di affitto inizia quando il cliente esegue **addRef** e termina quando il cliente esegue **removeRef** e comunque non oltre **T** unità di tempo
  - È responsabilità del cliente rinnovare l'affitto prima che scada
  - A causa di guasti di comunicazione che impediscono al cliente di rinnovare il lease, l'oggetto remoto può "sparire" al cliente
  - Quando l'affitto scade, il server considera il riferimento remoto non più valido e lo rimuove
  - La durata del lease è controllata da **java.rmi.dgc.leaseValue**