Java Remote Method Invocation (Java RMI)

Advanced Computer Programming

Prof. Luigi De Simone

Argomenti

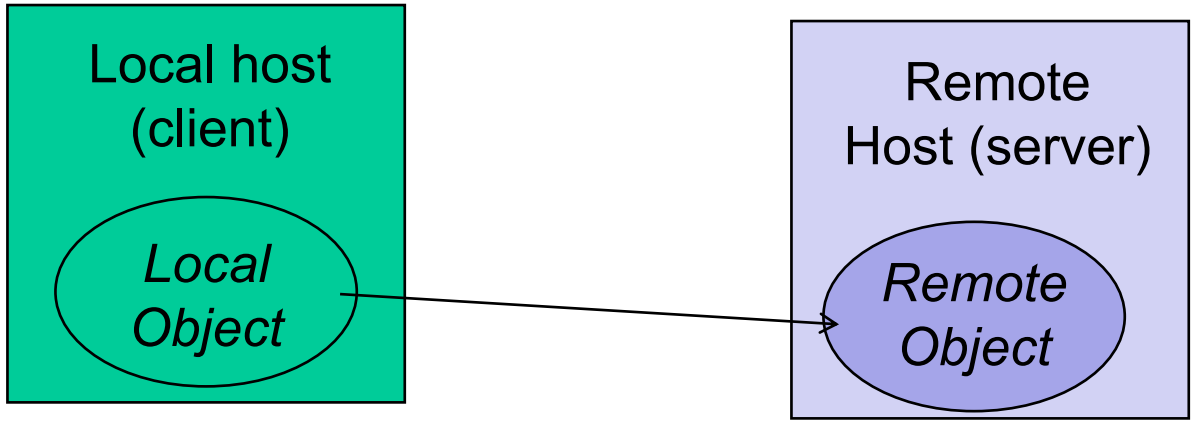
- Java Remote Method Invocation (Java RMI)
  - Concetti introduttivi;
  - Scrittura di un programma Java RMI;
    - RMI Registry;
    - Stub e Skeleton RMI;
  - Passaggio dei parametri;
  - Serializzazione e codebase;
  - Architettura;
  - Policy file e Security Manager;
  - Callback distribuita.

Riferimenti:

- Tutorial RMI: http://docs.oracle.com/javase/tutorial/rmi/
- Materiale didattico

Java RMI

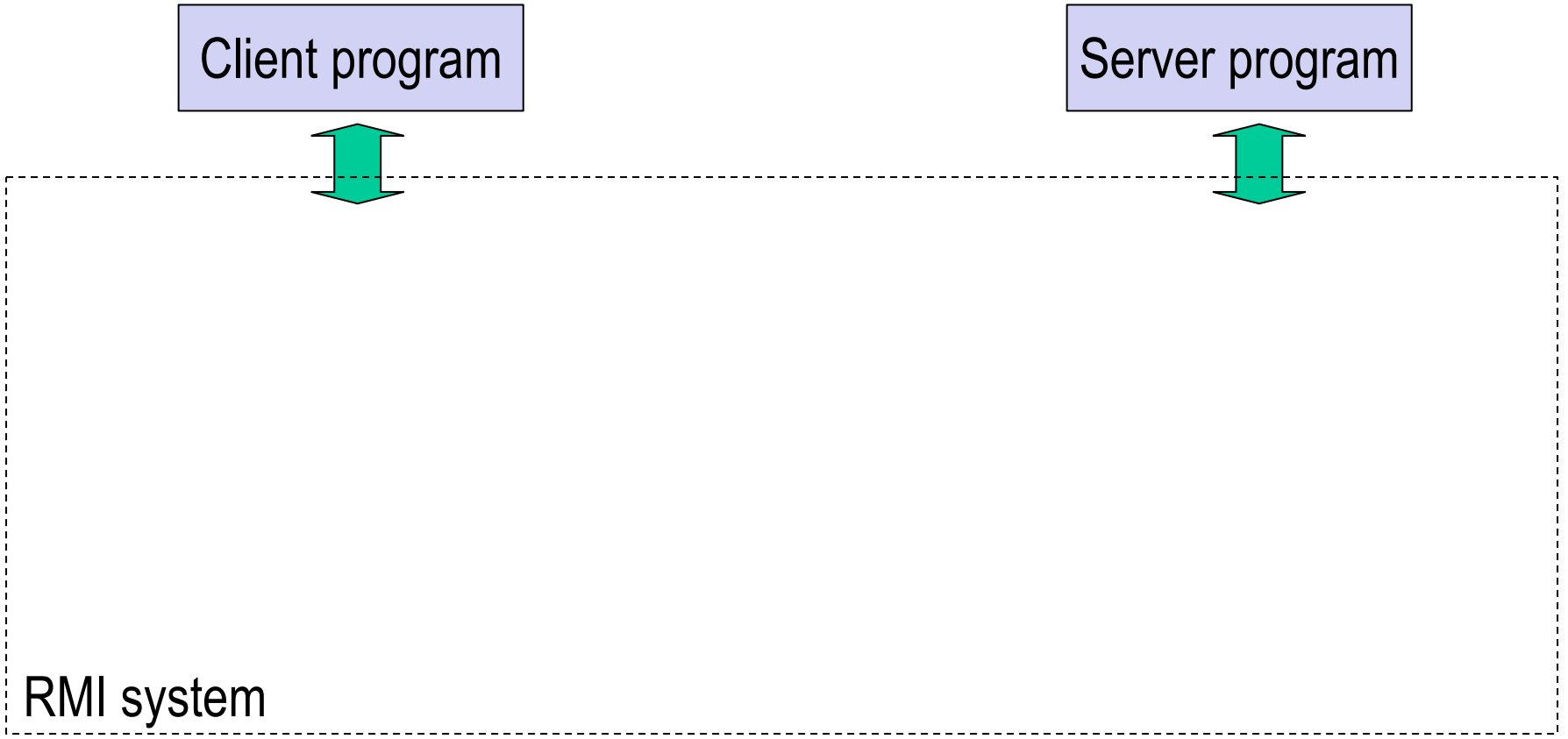
- Implementazione Java dei meccanismi di comunicazione tra oggetti in ambiente distribuito, ovvero un middleware ad oggetti distribuiti (DOM)
- Java RMI consente di distribuire oggetti sui nodi di una rete ed invocarne i metodi remotamente:
  - Il metodo invocato risiede in uno **spazio di indirizzamento remoto**;
  - La richiesta di invocazione di un metodo da parte di un client consiste in un **messaggio** al server che *gestisce* l'oggetto "reale" (detto anche oggetto **servente**);
  - Viene invocato un metodo sull'oggetto **lato server,** ed il risultato è restituito al client con un messaggio di risposta.



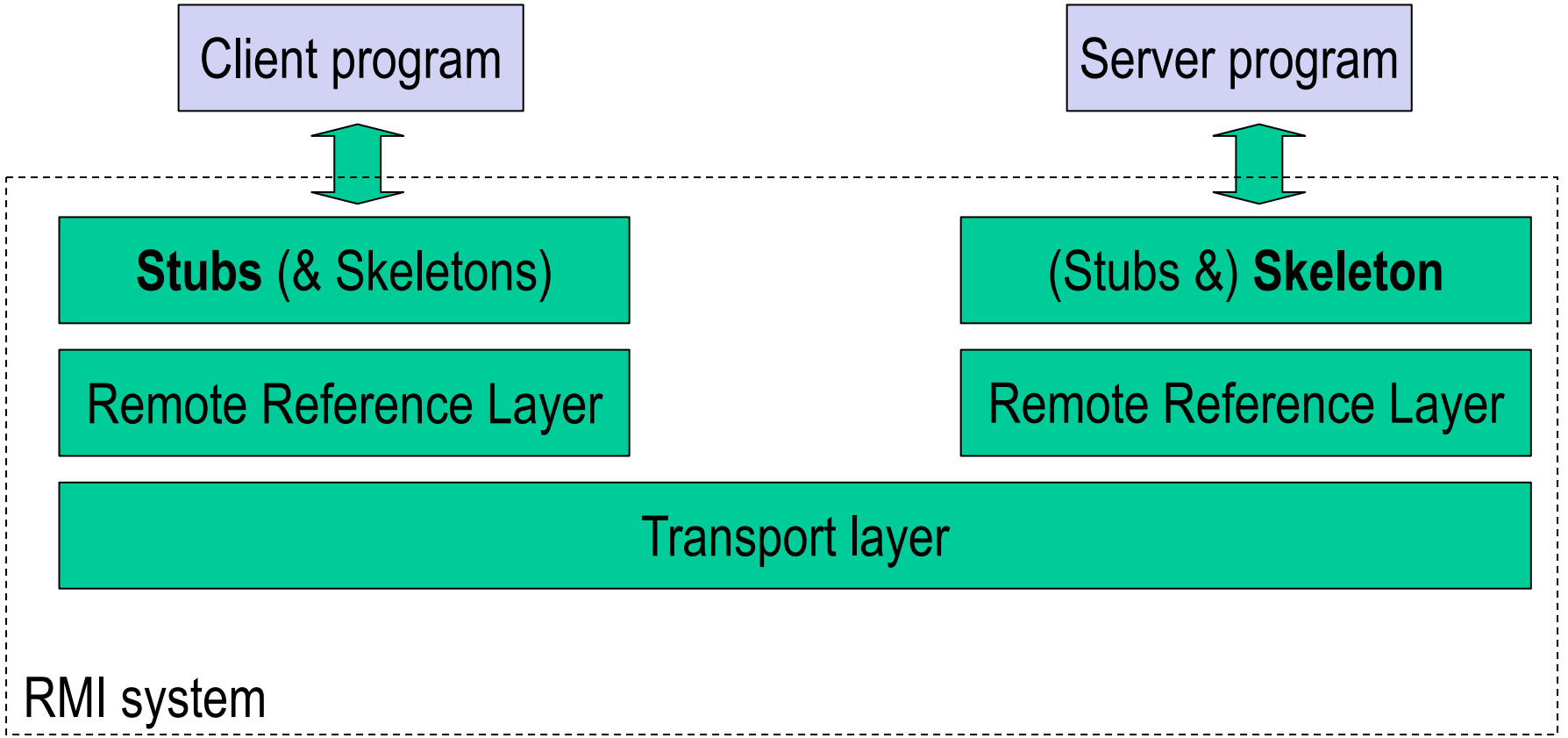
Caratteristiche Java RMI

- Solo **Java-to-Java**:
  - in un sistema multi-linguaggio (come ad esempio CORBA), si rende invece necessario un IDL ed il relativo mapping sul linguaggio utilizzato per l'implementazione.
- E' un meccanismo di **tipo client-server**;
- Fornisce una API di alto livello;
- Proprietà di **trasparenza rispetto alla locazione**.

Organizzazione architetturale



Organizzazione architetturale



Definizione di una interfaccia Remote

- Esempio: interfaccia di un servizio Counter (contatore remoto).

```
import java.rmi.*;

public interface Counter extends Remote {

        public int getCount() throws RemoteException;
        public void increment() throws RemoteException;

}
```

Definizione di una interfaccia Remote

- L'interfaccia estende java.rmi.Remote, che indica che stiamo **definendo un servizio remoto**.
- RemoteException:
  - in ogni istante sono possibili fallimenti dovuti a problemi di comunicazione, rete, malfunzionamenti lato server;
  - ogni eccezione dovuta ai problemi citati dovrebbe essere gestita durante l'invocazione dei servizi remoti;
  - caratteristica necessaria per i servizi RMI.

Implementazione del servizio remoto

- **Soluzione 1 (ereditarietà):** "`extends UnicastRemoteObject implements Counter`"

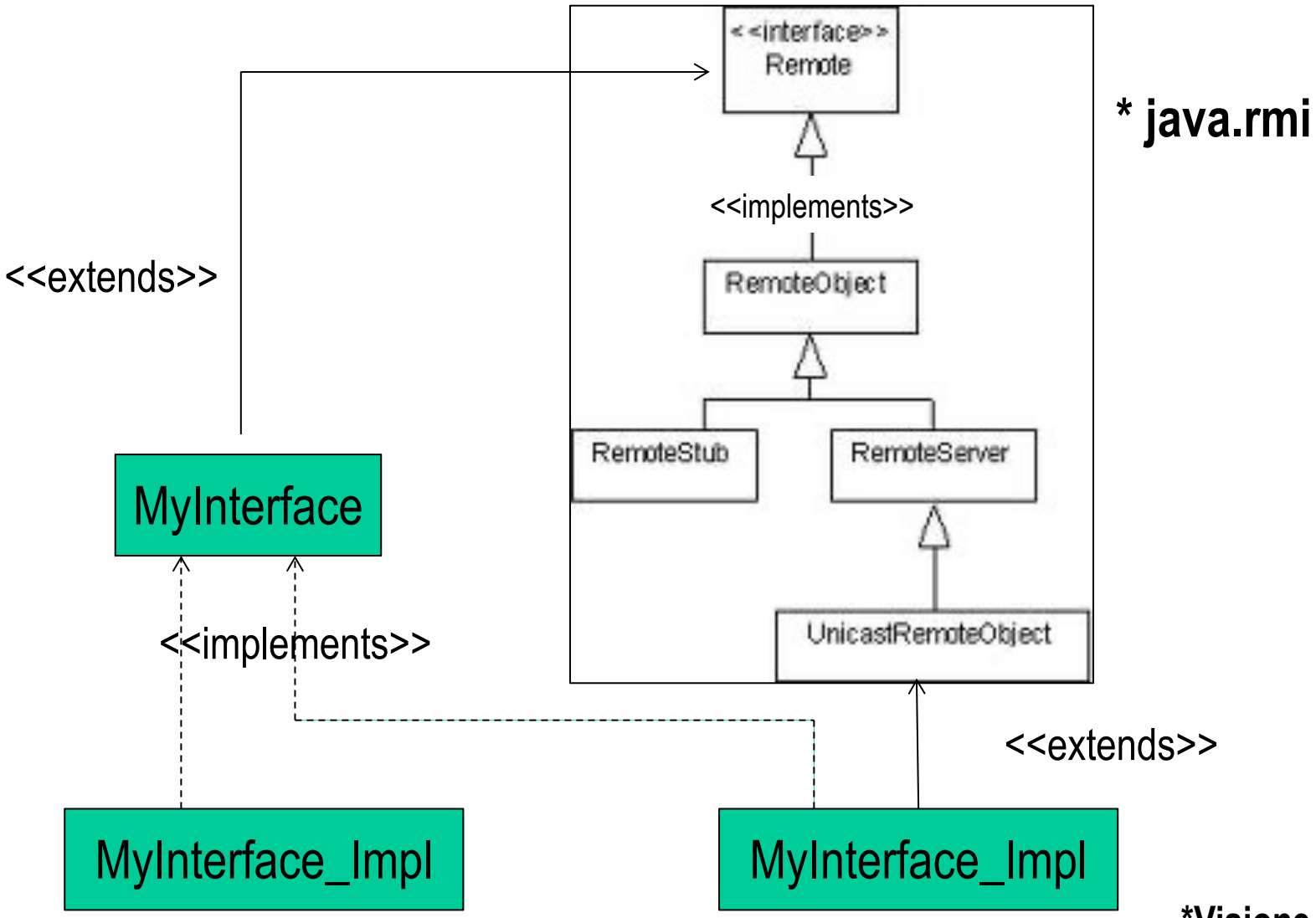
```
import java.rmi.*;
Import java.rmi.server.*;

public class CounterImpl  extends UnicastRemoteObject
                implements Counter {

        private int count;

        public CounterImpl() throws RemoteException{

                counter =0;
        }

        public int getCount() throws RemoteException{

                return count;
        }

        //other methods
}
```

Implementazione del servizio remoto

- **Soluzione 2 (delega)**: "`implements Counter`"

```
import java.rmi.*;
public class CounterImpl implements Counter {
        private int count;
        public int getCount() throws RemoteException{
                return count;
        }
        //other methods
}
```

Class diagram



*Visione semplificata. Si faccia riferimento alla javadoc per l'elenco delle classi completo

Avviare il servizio remoto

- Al fine di poter utilizzare il servizio, è necessario creare e, se necessario ***esportare*** l'oggetto servente sull'infrastruttura RMI
- **Soluzione per ereditarietà** (crea oggetto servente)

```
//nel caso della soluzione 1: il servizio è esportato
automaticamente su RMI, e messo in ascolto di richieste remote;

Counter counter = new CounterImpl();
```

- **Soluzione per delega** (crea ed esporta oggetto servente)

```
//nel caso della soluzione 2: sebbene il servizio implementi
Remote, è necessario esportarlo in maniera esplicita su RMI;


Counter counter = new CounterImpl();
Counter counterStub = (Counter)UnicastRemoteObject
                      .exportObject(counter, 0);
```

Contattare un oggetto remoto

- Affinché un client possa contattare un oggetto remoto è necessario che esso disponga del **riferimento remoto** dell'oggetto e, se necessario, del codice della classe stub ad esso associata:
  - una possibile soluzione "trivial": salvare lo stub prodotto dal server su un file e copiare la classe stub lato client.

- Java RMI utilizza un registro speciale chiamato RMI Registry con cui è possibile individuare e gestire la locazione degli oggetti distribuiti:
  - **Trasparenza rispetto alla locazione!**

RMI Registry

- Avviato l'oggetto remoto, esso viene collegato (operazione denominata *binding*) ad un **nome simbolico** nel registro RMI:
  - tipicamente tale operazione è effettuata dal main program server o, in generale, da chiunque voglia rendere disponibile un oggetto remoto

```
import java.rmi.registry.*;

//…

//NOTA: oggetto remoto tramite ereditarietà
// CounterImpl extends UnicastRemoteObject

Counter counter = new CounterImpl ();

Registry rmiRegistry = LocateRegistry.getRegistry();
rmiRegistry.rebind("mycounter", counter);
```

  - *i)* `getRegistry` restituisce un riferimento al registry (NOTA: l'RMI registry è esso stesso un oggetto RMI!) e, *ii)* si associa un nome simbolico (per es. "mycounter") al servizio remoto.

RMI Registry

- Il client del servizio utilizza il nome simbolico per ottenere il riferimento remoto dell'oggetto (operazione denominata *lookup*)

```
import java.rmi.registry.*;

//…

Registry rmiRegistry = LocateRegistry.getRegistry();
Counter counter = (Counter)rmiRegistry.lookup("mycounter");

counter.increment(10);     //Invocazione remota!!!
```

- Come prima, si ottiene un riferimento al registry e si cerca il riferimento dell'oggetto remoto a partire dal nome simbolico *"mycounter"*:
  - a seguito di una **lookup**, quando necessario, **viene recuperato il codice della classe Stub**.

RMI Registry

- Il registro RMI funge da **repository centralizzato** per la gestione dei riferimenti degli oggetti remoti:
  - Non si occupa delle invocazioni remote stesse!
- E' un servizio della piattaforma RMI che va **avviato prima di lanciare le applicazioni** che ne fanno uso:
  - è un processo del sistema operativo;
  - utilizza la porta di default **1099**;
  - è possibile utilizzare una porta differente, specificandola all'avvio del registro (NOTA: alla getRegistry andrà passata tale porta al fine di poter individuare il registro).

  Avviare il registro RMI da terminale:

  Windows: rmiregistry (assumendo che java/bin sia nel PATH)

  Unix/MacOS: rmiregistry

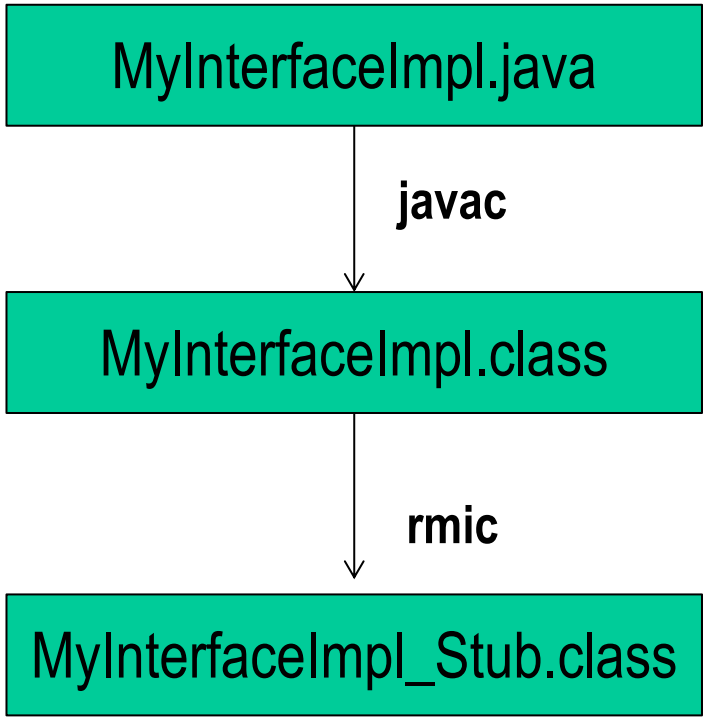
Stub e skeleton

- Gli Stub possono essere generati:
  - dinamicamente durante l'esecuzione del programma;
  - automaticamente tramite il compilatore **`rmic`**; **tuttavia, gli stub generati con rmic sono deprecati (obsoleti).**

- Gli skeleton non sono necessari in RMI (da Java 6 in poi):
  - gli stub RMI inviano, come parte dei parametri dell'invocazione remota, tutte le informazioni che servono ad identificare l'oggetto remoto (e relativo metodo) da invocare lato server;
  - gli skeleton sono sostituiti da un **dispatcher generico** lato server che effettua l'*upcall* sugli oggetti serventi (utilizzo della **reflection**, come indicato più avanti).

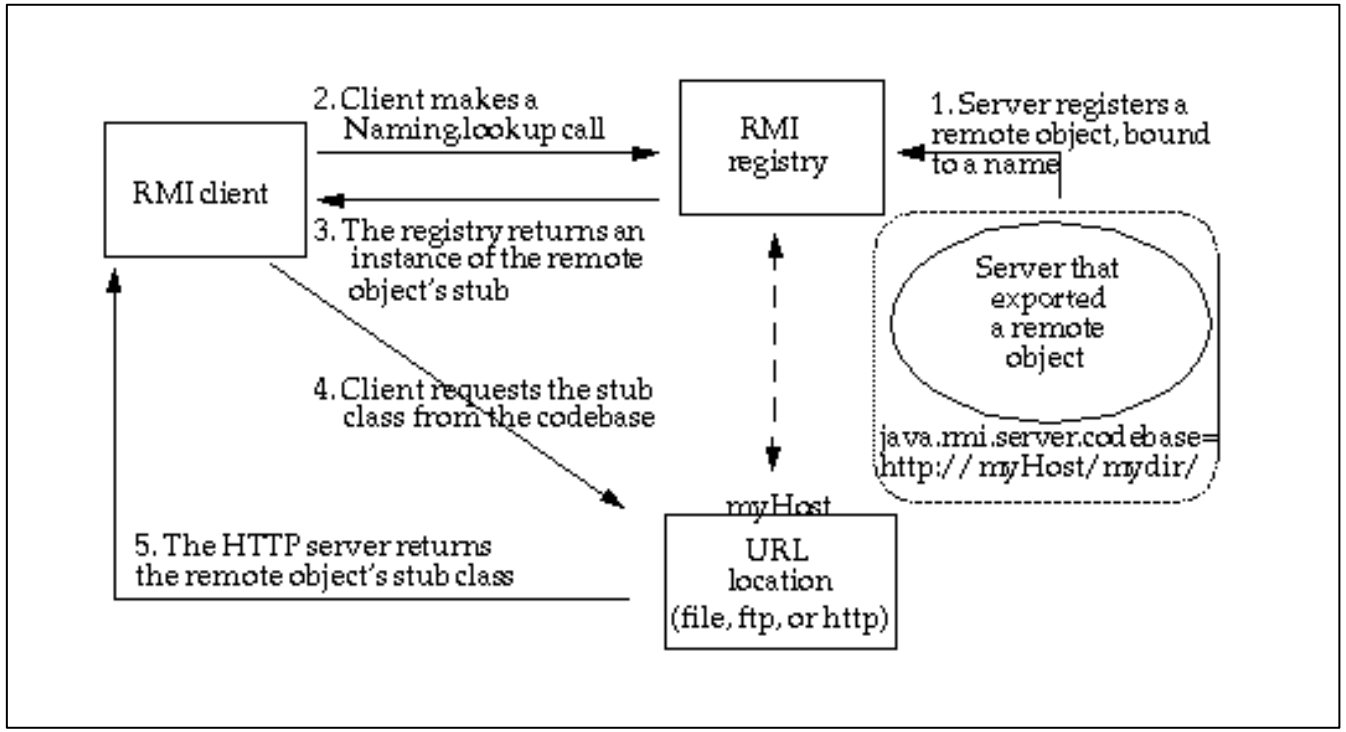
Nota su rmic

- **`rmic`** è un compilatore speciale disponibile con il JDK.
  - Legge il file .class contente l'implementazione del servizio;
  - Produce i file _Skel.class e _Stub.class;
  - Stub generati con rmic potrebbero essere comunque necessari per motivi di **backward compatibity;**



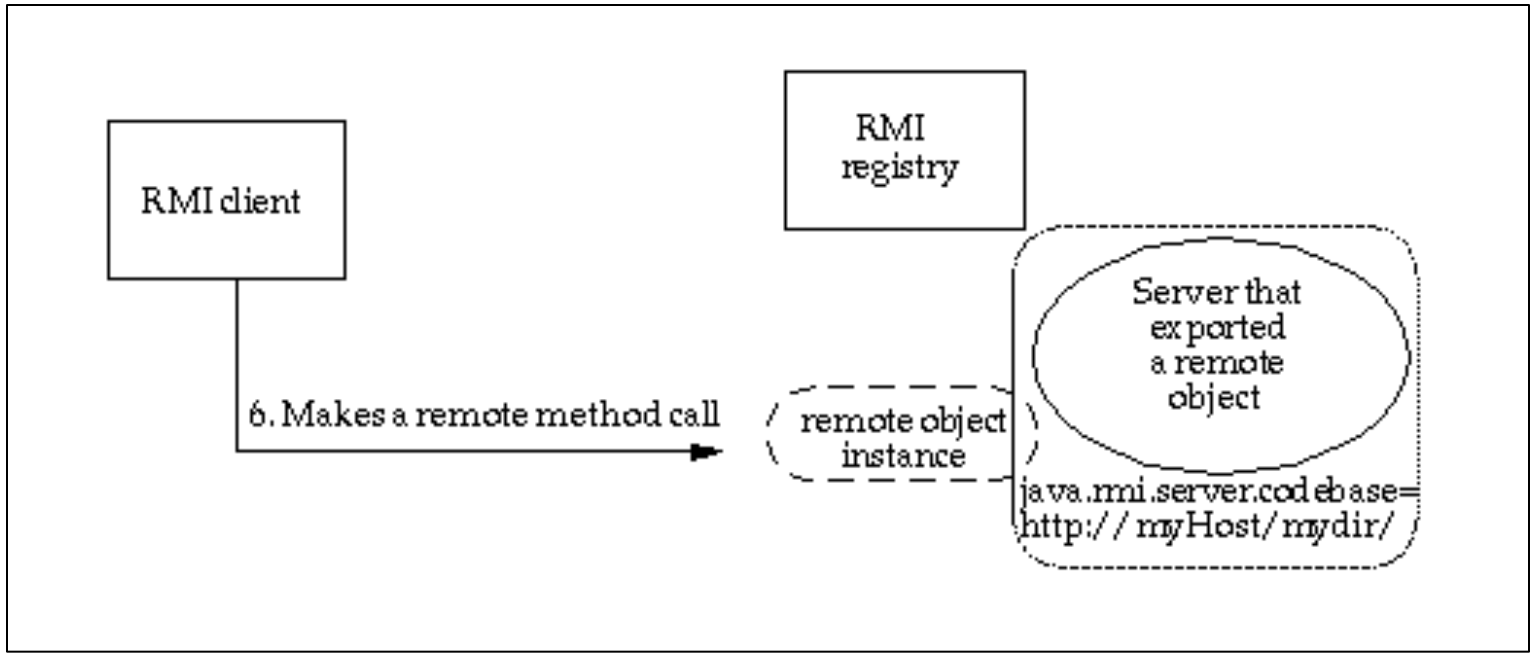
Funzionamento (1/2)

- **Il server inserisce il servizio nel registro RMI** (**1**). Ad essere registrato è il riferimento remoto, contenente le informazioni che i client dovranno utilizzare per contattare il server (per es., IP/porta della socket);
- Il client **ottiene il riferimento remoto** effettuando una *lookup* sul registro (**2, 3**);
- Il client **scarica il codice della classe stub** (**4, 5**);



Funzionamento (2/2)

- Ottenuti il riferimento remoto e il codice dello stub, **il client può effettuare l'invocazione remota** (**6**):
  - Apertura della socket verso il server;
  - Marshalling dei parametri;
  - Invio della richiesta.



Passaggio dei parametri e risultati

- In Java RMI i parametri di un metodo sono parametri di ***input***, mentre quello di ritorno è di ***output***:
  - Ogni oggetto che implementi l'interfaccia Serializable può essere utilizzato come **argomento** o **risultato** di **metodo remoto**;
  - Un oggetto **serializzato** è accompagnato dalla *locazione* del bytecode della classe di appartenenza (una **URL**, come specificato in seguito) al fine di consentirne il download da parte del ricevente;
  - **I tipi primitivi e gli oggetti remoti sono serializzabili**.
- I tipi primitivi sono passati per valore
- **Passaggio degli oggetti remoti:**
  - Essi sono passati come **riferimenti a oggetti remoti**;
  - Quando viene ricevuto un riferimento ad un oggetto remoto, esso può essere utilizzato per effettuare una RMI.

Passaggio dei parametri e risultati

- **Passaggio degli <u>oggetti non-remoti</u>:**
  - Tutti gli oggetti **Serializzabili** non-remoti sono copiati e passati per **valore**;
  - Quando un oggetto è passato per valore, **un nuovo oggetto è creato nel processo ricevente;**
  - I metodi di questo oggetto possono essere quindi invocati localmente dal ricevente ed, eventualmente, causare una modifica di stato rispetto alla copia nel processo mittente.

Serializzazione di un oggetto

- **Serializzare** consiste nel trasformare un oggetto in una forma adatta alla memorizzazione su disco o trasmissione tramite messaggio su rete (per es., argomento o risultato di una RMI):
  - analogamente "**Deserializzare**" consiste nel ricostruire un oggetto a partire dalla sua forma serializzata.

- Le procedure di **serializzazione** e **deserializzazione** sono una componente delle operazioni di *marshalling* e *unmarshalling* dei parametri.

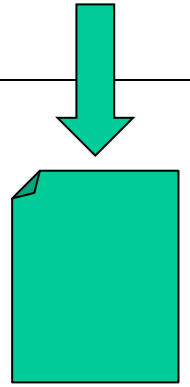
- Affinché si possa trasferire un oggetto -come parametro o risultato di una RMI-, è necessario che la relativa classe di appartenenza implementi **Serializable** (nota: fa parte del package **Java IO**):
  - E' un'interfaccia *marker*, non richiede l'implementazione di particolari metodi;
  - Non tutte le classi sono Serializzabili, per es., Thread.

Serializzazione di un oggetto

- Si assume che il processo ricevente –ossia quello destinato ad effettuare la *deserializzazione*– non abbia informazioni *a priori* sul tipo dell'oggetto serializzato:
  - ne consegue che la **forma serializzata** debba includere delle **informazioni** riguardo la classe di appartenenza dell'oggetto, e che consentano al ricevente di deserializzare correttamente l'oggetto;
  - tali informazioni sono: **nome della classe** ed un **version number**.
- Un oggetto in Java può avere riferimenti ad altri oggetti:
  - quando un oggetto è serializzato, tutti gli oggetti da esso referenziato sono a loro volta serializzati;
  - variabili che non devono essere serializzate sono dichiarate ***transient***.

Esempio: class Person

```
public class Person implements Serializable {
        private String name;
        private String place;
        private int year;

        public Person(String aName, String aPlace, int aYear) {
                name = aName;
                place = aPlace;
                year = aYear;
        }

        // followed by other methods for accessing the instance
        //variables
        // …
}
```

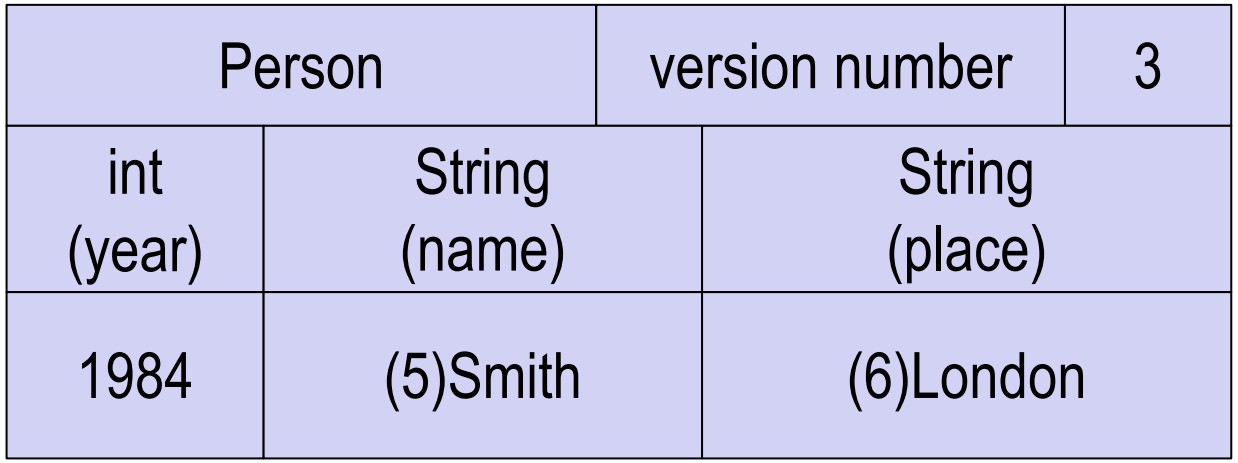


Person.class

Serializzazione di un oggetto Person

```
Person p = new Person("Smith", "London", 1984);
```

- Oggetto serializzato (visione semplificata):



- Oltre all'oggetto serializzato, il receiver necessita del file con il **codice** della classe (**Person.class**, nell'esempio):
  - pertanto, oltre a **nome della classe** e **version number**, l'oggetto serializzato è accompagnato dalla **locazione** (una URL denominata **codebase**) del bytecode della classe di appartenenza da cui il receiver potrà scaricare la classe.

Version number

- Come specificato in precedenza, un oggetto serializzato è caratterizzato anche da un **version number** della classe di appartenenza*:*
  - è i) settato dal programmatore oppure ii) generato automaticamente (in funzione del nome della classe, delle variabili membro, dei metodi e delle interfacce);
  - il version number cambia quando sono effettuate modifiche sostanziali al codice della classe.

- **Il processo che deserializza** un oggetto **verifica** che si trovi in possesso della **versione corretta della classe**:
  - è possibile determinare il serial version number con l'utility serialver

```
protected static final long serialVersionUID=…;
```

Codebase

- La codebase è identificata da una **URL** (Uniform Resource Locator). Una URL è una stringa di testo che identifica una risorsa.
  Formato URL: *protocol://host/location*
  - *protocol* è il protocollo di accesso (http, file, …)
  - *host* nome della macchina o indirizzo
  - *location* file o posizione della risorsa

- Come anticipato, la conoscenza della **codebase** è necessaria ad un **receiver** (durante le operazioni di deserializzazione) per recuperare il codice della classe dell'oggetto, serializzato, ricevuto:
  - in assenza di informazioni sulla codebase, il receiver proverà a cercare la classe sul file system locale ai path specificati dalla variabile **CLASSPATH**.

Codebase

- La **codebase** deve essere specificata dal programma ***sender*** di un oggetto non-remoto o di un riferimento remoto.
- La codebase si imposta tramite la proprietà java.rmi.server.codebase all'avvio del programma. Per es.:

```
java -Djava.rmi.server.codebase=file:///PATH-TO-STUB/ server.CounterServer
```

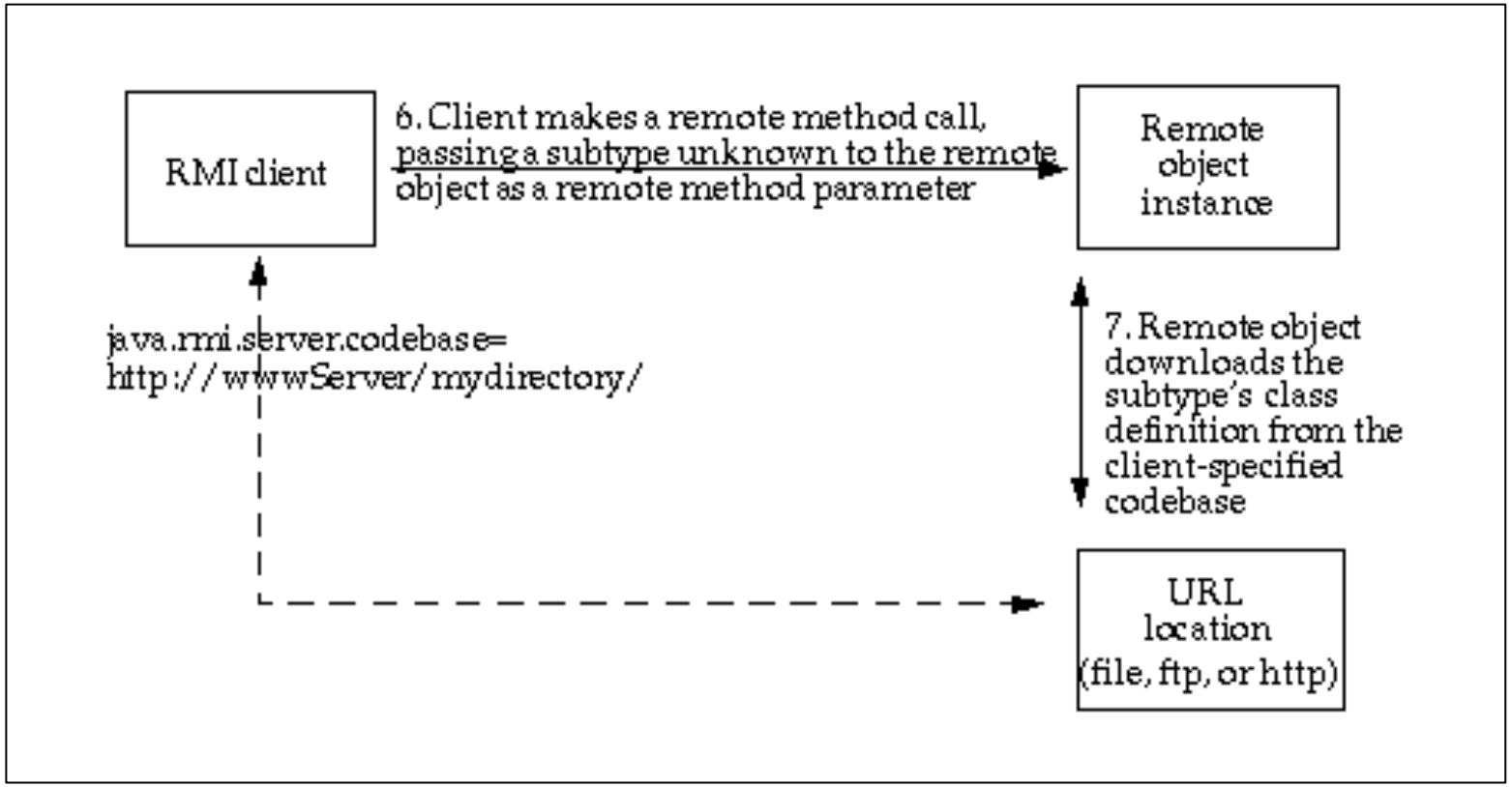
  La URL **file://** specifica la posizione nel file system del codice dello stub (NOTA: non dimenticare di inserire il carattere '/' finale)

- Una codebase non risiede necessariamente sul file system locale:
  - può essere una qualsiasi URL valida che specifica la posizione in rete da cui può essere scaricato il bytecode di una classe;

    `-Djava.rmi.server.codebase=http://webvector/export/`
  - Esempi di URL valide: http:// o ftp://

Codebase

- Per esempio, la codebase è necessaria:
  - ad un **client** (o all'RMI Registry) per costruire dinamicamente il proxy associato ad un riferimento remoto;
  - ad un **server** che riceve come argomento di una Java RMI un oggetto non-remoto (esempio in figura sottostante).



Note su RMI registry e codebase

- Il download delle classi dalla locazione specificata da un endpoint RMI è regolato dalla proprietà **java.rmi.server.useCodebaseOnly***
  - in diverse versioni del JDK il valore di default di tale proprietà è true (ossia le classi non possono essere scaricate dalla location specificata dall'endpoint RMI).
  - Per esempio, nel caso di rmiregistry la proprietà **useCodebaseOnly** è specificata all'avvio, come segue:

```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

- Esempi di specifica della codebase su Windows (URL file):

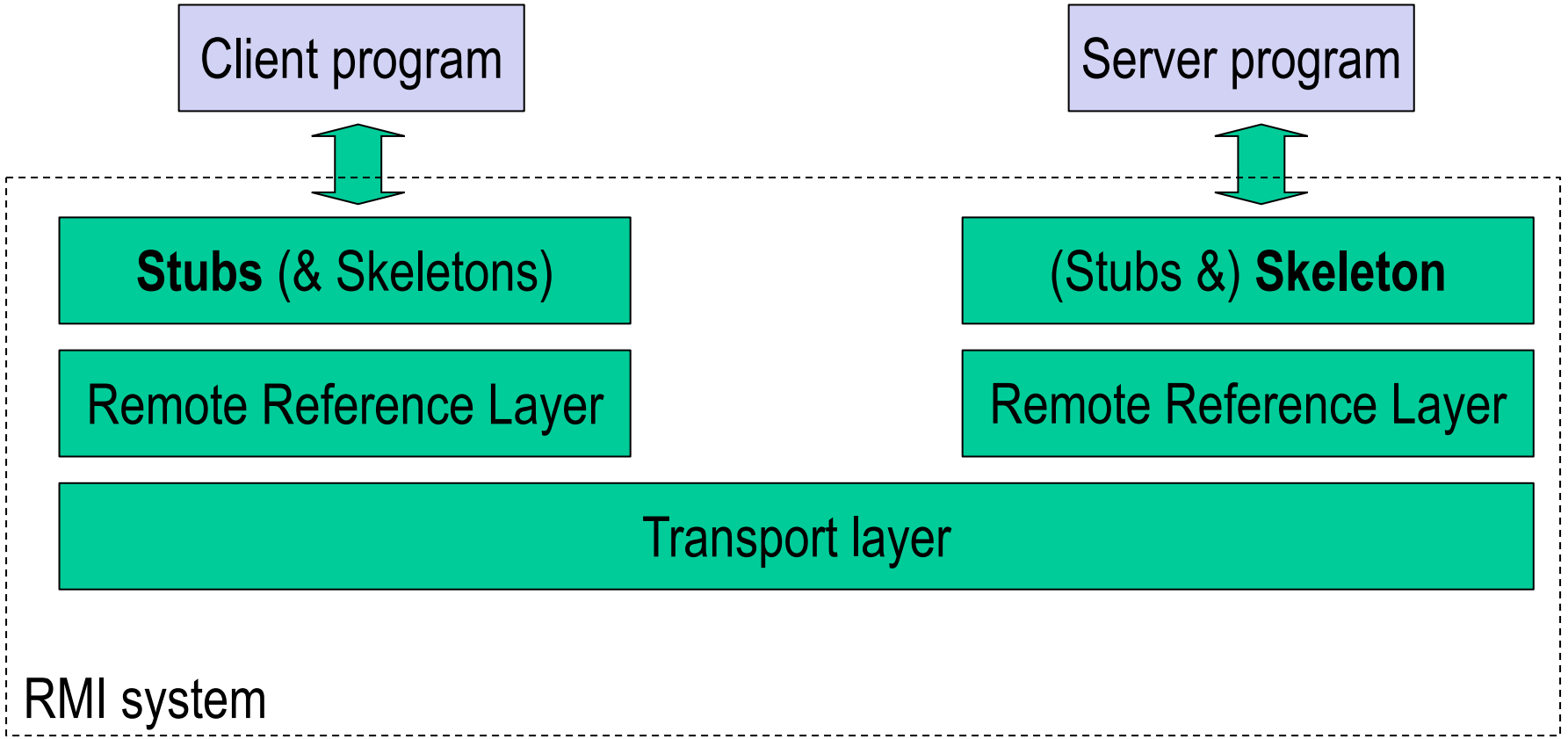
```
java -Djava.rmi.server.codebase=file:/c:\workspace\rmicounter\bin/
java -Djava.rmi.server.codebase=file:/c:\Documents%20and%20Settings\Administrator\workspace\rmicounter\bin/
```

*http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/enhancements-7.html

Cenni implementativi

Cenni implementativi

- Architettura di un sistema RMI



Gestione delle connessioni

- Il **transport layer**, è responsabile delle **connessioni** verso lo spazio di indirizzamento remoto:
  - la comunicazione è implementata sul protocollo TCP/IP;
- Rimane in attesa di connessioni in ingresso.
- **Tipologie di oggetti remoti:**
  - Unicast point-to-point: prima che l'oggetto possa essere utilizzato, è necessario istanziarlo ed esportarlo su RMI (JDK 1.1);
  - Attivabili: all'invocazione di un metodo remoto, RMI determina se il servizio è dormiente. Se si, RMI istanzia l'oggetto recuperandone lo stato dal disco (Java 2).
  - Oggetti multicast: il proxy inoltra simultaneamente la richiesta a più implementazioni remote e restituisce la prima risposta ricevuta.

Gestione delle connessioni

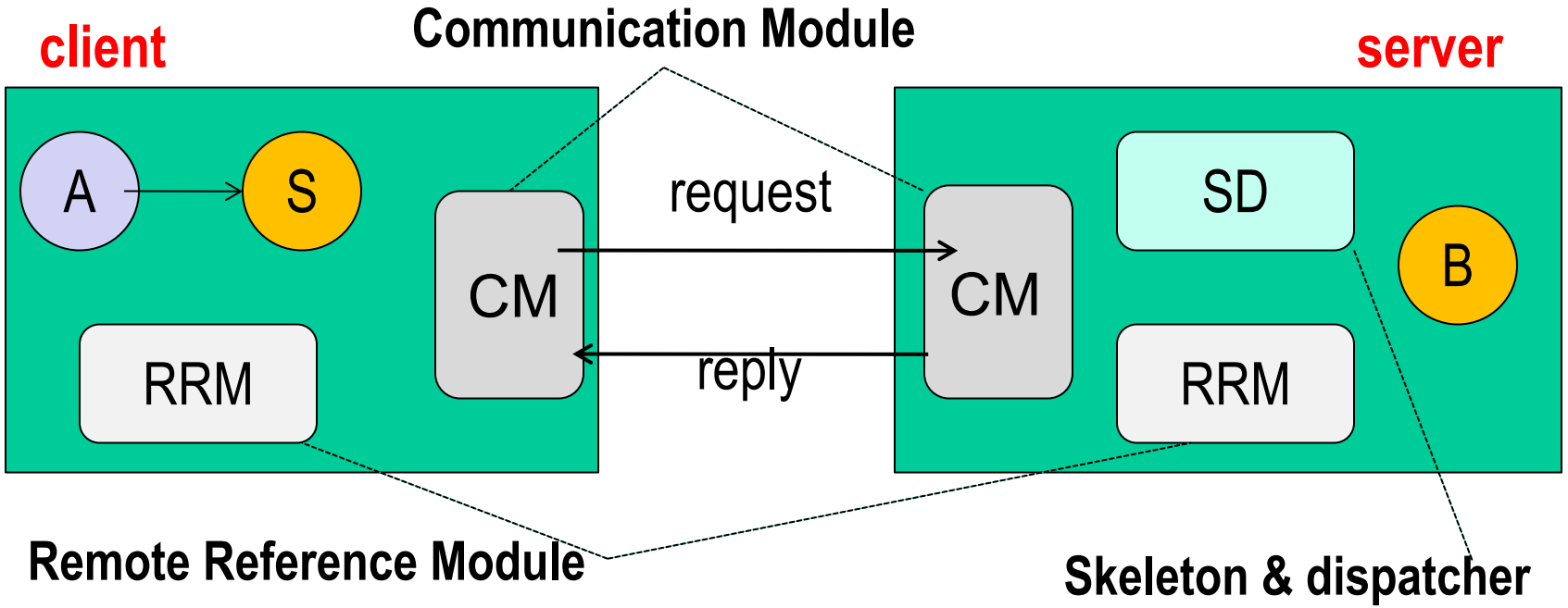
- **Più client** possono ottenere uno stub che *punta* al medesimo oggetto remoto
- Quando un client si connette, il **server** avvia un nuovo thread per gestire l'invocazione remota e ritorna in attesa di nuove connessioni:
  - L'oggetto remoto è acceduto in concorrenza;
  - Quando necessario, è opportuno **sincronizzare gli accessi** sull'oggetto lato server.

- **La specifica generica RMI** per la gestione delle **invocazioni** remote prevede **due alternative**:
  - **Single-op**: ogni call RMI è effettuata su una socket che poi viene chiusa.
  - **Stream**: più chiamate RMI sulla stessa socket, una di seguito all'altra.

Gestione delle connessioni

- **Java RMI di *Sun Microsystems* utilizza il meccanismo di stream**
  - Uno **stub lato client** utilizzerà **una singola socket** verso il server **per tutte le invocazioni remote**.
- **Stub differenti** nella stessa JVM (o in differenti JVM sulla stessa macchina) **avranno socket differenti**
  - Ogni volta che si ottiene uno stub dall'RMI registry, verrà creata una nuova socket per comunicare con il server.
- Quindi, esiste una socket *per stub* **lato client**, e se il client ha più stub che puntano allo stesso oggetto remoto, ci saranno più socket aperte verso il server.
- Pertanto, ad un oggetto **`UnicastRemoteObject`** **lato server** sono associati più thread che effettuano delle invocazioni sul medesimo oggetto (un thread per socket).

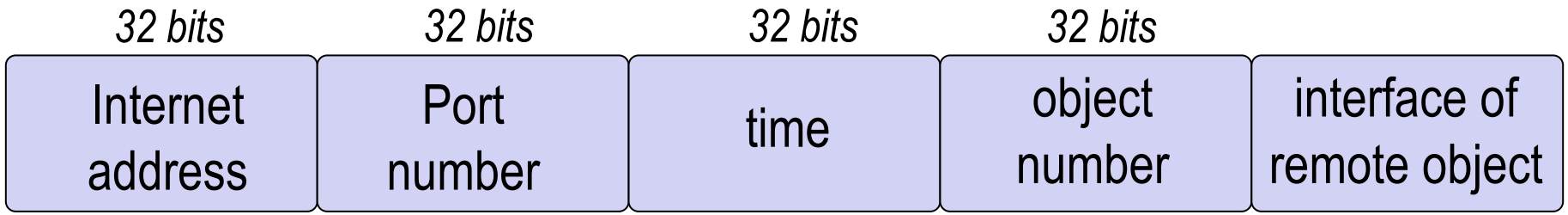
Implementazione di una RMI



- B è un oggetto *servente* (ed m un suo *metodo*); S lo stub lato client.
- Nel seguito
  - ***R*** è un **riferimento remoto (locale al client)**
  - ***r*** è un **riferimento locale** sul server dell'oggetto ***B*** **(locale al server)**

Riferimento remoto

- Un riferimento remoto (**remote object reference**) è un identificativo di un oggetto remoto all'interno di un sistema distribuito:
  - è incluso nei **messaggi di richiesta** per individuare l'oggetto da invocare;
  - le *remote references* possono essere passate come argomenti o restituite come risultato di una invocazione di metodo remoto.
- Sono generate in modo da essere **univoche** nello *spazio* e nel *tempo*:
  - quando un oggetto remoto è deallocato, la relativa *remote reference* non dovrebbe essere riutilizzata in quanto i client potrebbero ancora conservare il vecchio riferimento: un tentativo di invocazione di un oggetto remoto deallocato dovrebbe generare un errore;
  - per esempio, una *remote reference* può essere **generata** concatenando l'indirizzo dell'host e la porta con l'istante di creazione dell'oggetto ed un id numerico.



Remote Reference Module (RRM)

- Tale modulo ha la responsabilità di creare e gestire i riferimenti remoti.
- Ha il compito di tradurre un riferimento *remoto* **R** in un riferimento *locale* **r** e viceversa:
  - la traduzione è implementata mantenendo una **Remote Object Table**

Remote reference	Local reference
R1	r1
R2	r2
…	…

Remote Reference Module (RRM)

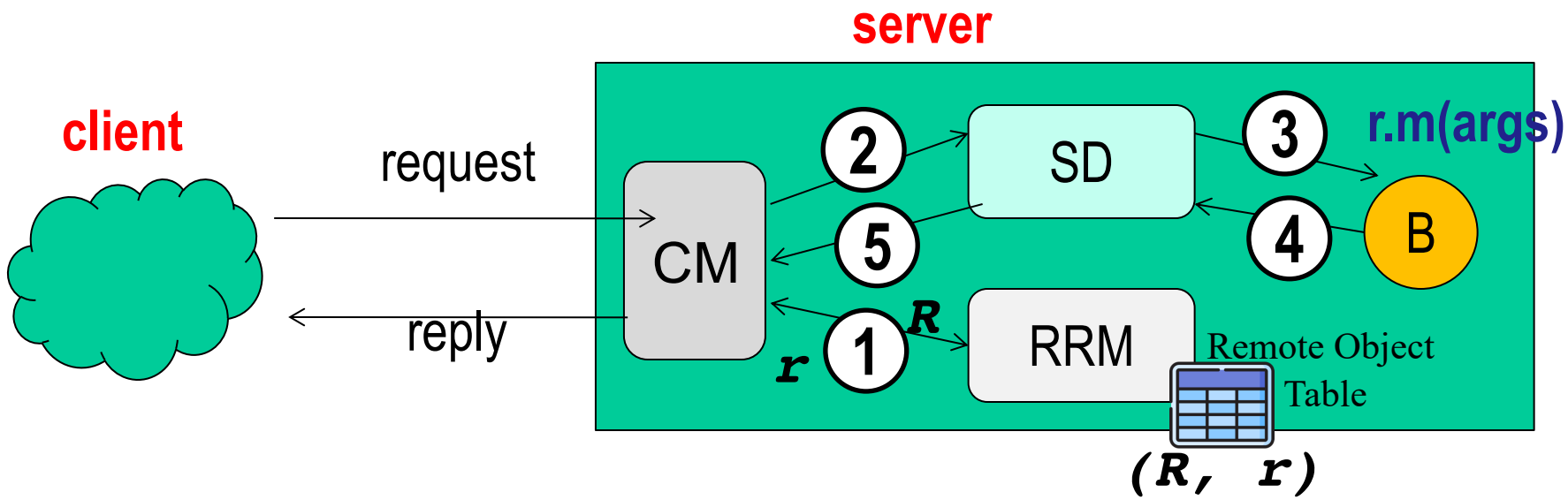
- Quando un oggetto remoto è passato come **argomento** o **risultato** per la prima volta, all'RRM è chiesto di creare un riferimento remoto, che è aggiunto alla *remote object table*;
- Quando un riferimento remoto R giunge ad un destinatario in un **messaggio di richiesta o risposta**, all'RRM è chiesto di restituire il riferimento locale r corrispondente ad R.
  - Per esempio, all'arrivo di un messaggio request *dal client al server*, l'RRM (e la relativa *remote object table*) è utilizzato per individuare l'oggetto server (lato server) da invocare.
- In generale, se un oggetto remoto è utilizzato per la prima volta, ed R non è presente nella Remote Object Table, RRM crea un riferimento remoto ed inserirà la entry ***(R, r)*** in tabella.

Communication module

- Implementa le operazioni di **request-reply**
- Responsabilità **lato client** del communication module:
  - Invia il messaggio di richiesta contente il *riferimento remoto* **R** dell'oggetto;
  - Semantica dell'invocazione: **at-most-once**
    - Non è possibile garantire che la procedura sia stata eseguita ma, **se è stata eseguita, lo è stata una volta sola**
- **Struttura del messaggio** request-reply

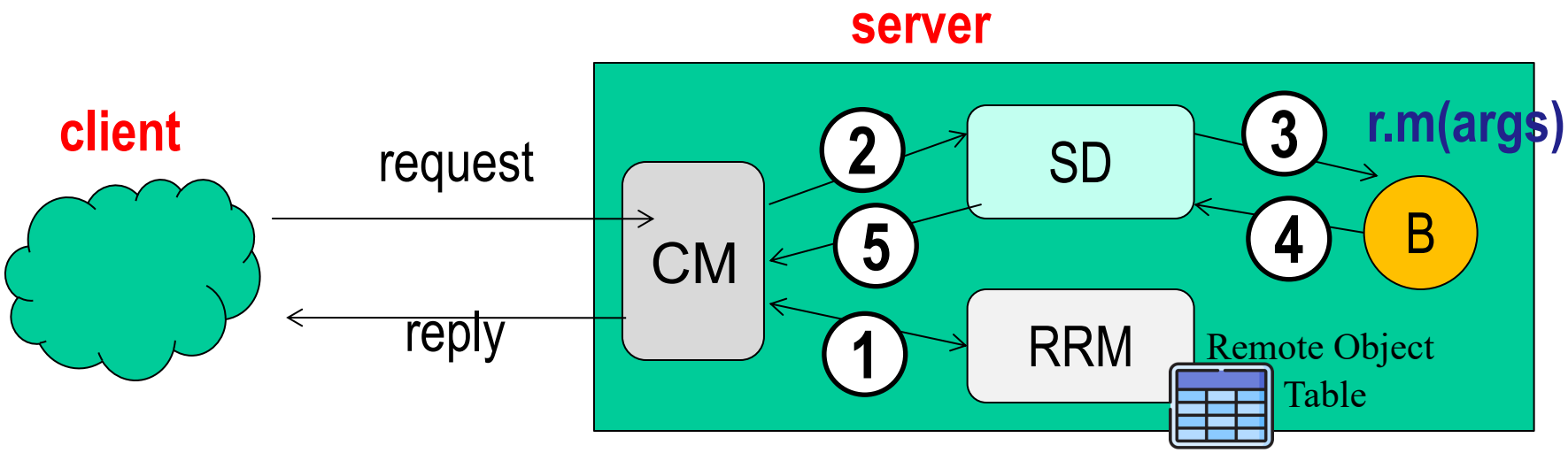
	TIPI
messageType	int (0=Request, 1=Reply)
requestID	int
RemoteReference	RemoteRef
operationId	int or operation name
arguments	//array di bytes

Communication module



- Responsabilità **lato server** del ***communication module (CM)***:
  - Il *communication module* riceve il messaggio di richiesta ed invia il riferimento remoto R al remote reference module (RRM) (**1**);
  - RRM accede alla **Remote Object Table** e restituisce al *communication module* il riferimento locale **r** corrispondente al riferimento remoto **R** (**1**);
  - Il *communication module* invia la coppia (messaggio di richiesta, r) allo Skeleton & Dispatcher (SD) (**2**).

Skeleton & dispatching



- Alla ricezione della coppia ***(messaggio di richiesta, r)*** da parte del *communication module*, lo Skeleton & Dispatcher (SD):
  - Determina il metodo da invocare, esegue l'unmarshalling dei parametri, ed effettua l'invocazione sull'oggetto servente B (**3**);
  - Attende il valore di ritorno (**4**);
  - Esegue il marshalling del valore di ritorno e consegna al *communication module* il messaggio di risposta (**5**);

Skeleton & dispatching

- Il modulo Skeleton & Dispatcher (SD) si basa sull'utilizzo della **reflection** (API **java.lang.reflect.***).
- La **reflection** consente a codice Java di **scoprire dinamicamente** informazioni riguardo ai **campi**, **metodi** e **costruttori delle classi caricate** nella **JVM**, e di costruire delle invocazioni di metodo.
- Il dispatcher utilizza le classi **Class** e **Method** della API reflect:

```
MyClass c = new MyClass();
Class cl = c.getClass();
Method m = cl.getDeclaredMethod("getValue", …);
m.invoke(c,…);        //invoca il metodo getValue sull'istanza c di MyClass
```

Security

- Una JVM può scaricare bytecode da una codebase specificata **da una qualsiasi URL valida** (file system locale, web server, etc).
- Se non in casi eccezionali (per es. la codebase specificata dalla JVM remota coincide con il *CLASSPATH* dell'applicazione che intende effettuare un download), bisogna gestire esplicitamente le policy e restrizioni connesse alle operazioni di download.

Security

- Un **policy file** contiene le informazioni sui permessi necessari all'applicazione Java per funzionate correttamente
- Consiste in una serie di ***grant* statements**

```
grant [signedBy Name] [codeBase URL] {
    //list of permissions
   };
```

"ogni classe che è stata scaricata dalla codebase URL *firmata digitalmente da* Name*, ha i seguenti permessi"*

Esempi

- **Esempio #1**

```
grant {
    permission
  java.security.AllPermission;
};
```

- **Esempio #2**

```
grant codeBase "file://d:/classes" {
  permission java.net.SocketPermission ":1024-",
  "accept, connect, listen, resolve";
  permission java.io.FilePermission "<<ALL FILES>>", "read";
};
```

Security - RMISecurityManager

- L'applicazione dovrà **assegnare un SecurityManager alla JVM**, che leggerà le policies e permetterà alla JVM di eseguire le operazioni richieste

```
if (System.getSecurityManager() == null)
    System.setSecurityManager ( new RMISecurityManager() );
```

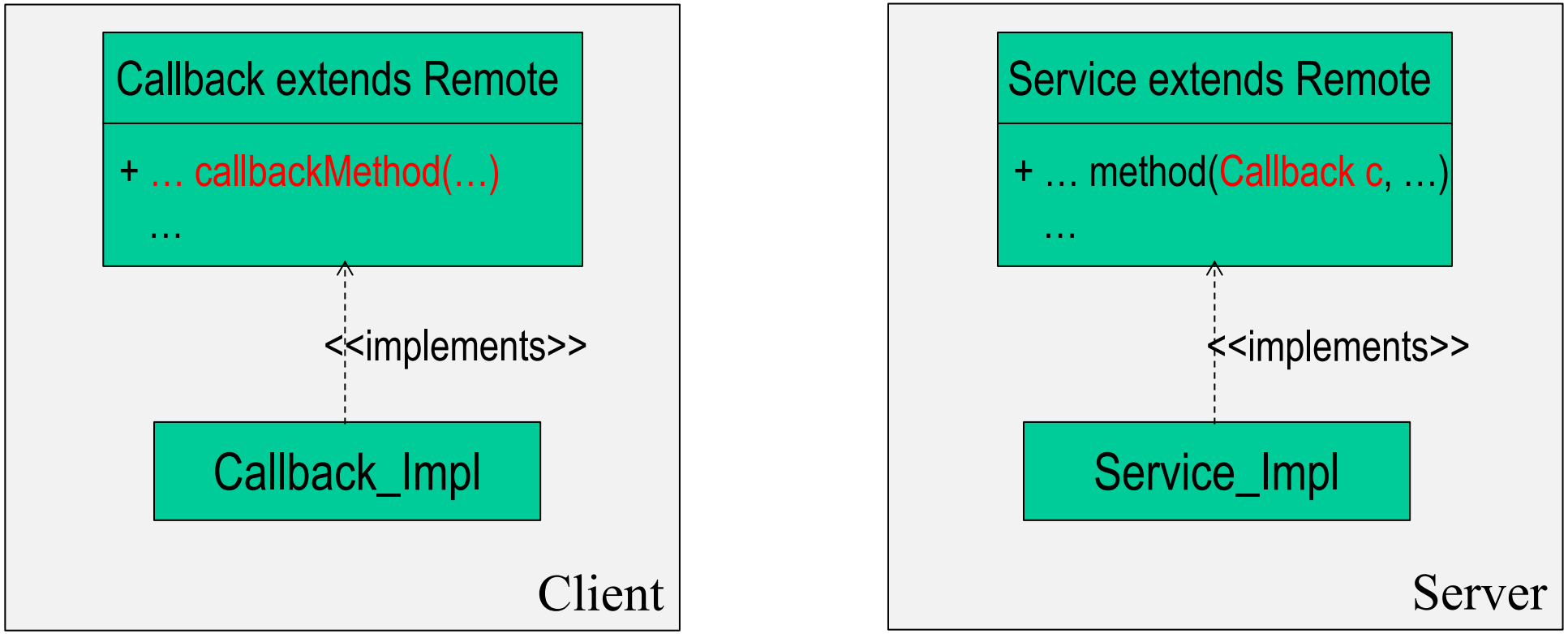
- **All'avvio dell'applicazione** verrà specificato il file contenente le policy che sarà letto dal security manager.

```
java -Djava.security.policy=myjava.policy client.Client
```

Callback

- L'idea alla base di una **callback** è che invece di avere dei client a fare "polling" sul server per verificare l'occorrenza di un determinato evento, sia il server stesso ad informare i client dell'occorrenza dell'evento.
  - **Il termine *callback* indica una invocazione dal server al client.**
- Implementazione di una **callback distribuita in Java RMI**:
  1. Il client crea un oggetto remoto (***callback object***) che implementa un'interfaccia contenente un metodo invocabile dal server;
  2. Il server fornisce un metodo che consente ai client di comunicare il riferimento remoto del *callback object* (il server memorizza i riferimenti in una lista);
  3. Quando si **verifica l'evento** di interesse, il server invoca il **callback object**.
- Esempi di utilizzo:
  - Notifica server→client (per es., **pattern observer**);
  - **Comunicazione asincrona.**

Schema generale in Java RMI



- Lo schema di comunicazione prevede **un'interfaccia `Callback`**, ed un oggetto remoto (istanza di **`Callback_Impl`**) avviato **lato client**, ed *invocabile* dal server
  - il client invierà il riferimento all'oggetto **`Callback`** invocando un metodo remoto di **`Service`**;
  - Il server potrà contattare il client, invocando remotamente un metodo di callback (ossia **`callbackMethod`** nell'esempio)

Funzionamento