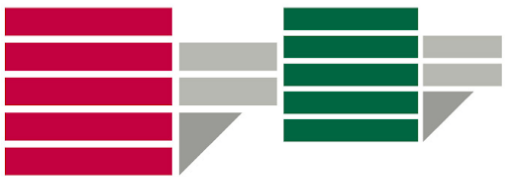


# Java RMI

UNIVERSITÀ DELLA CALABRIA



DIMES - Dipartimento di INGEGNERIA INFORMATICA  
MODELLISTICA, ELETTRONICA E SISTEMISTICA

Ing. Ludovica Sacco  
DIMES – UNICAL - 87036 Rende(CS) - Italy  
Email: [l.sacco@dimes.unical.it](mailto:l.sacco@dimes.unical.it)

# Paradigma di Programmazione degli Oggetti Distribuiti

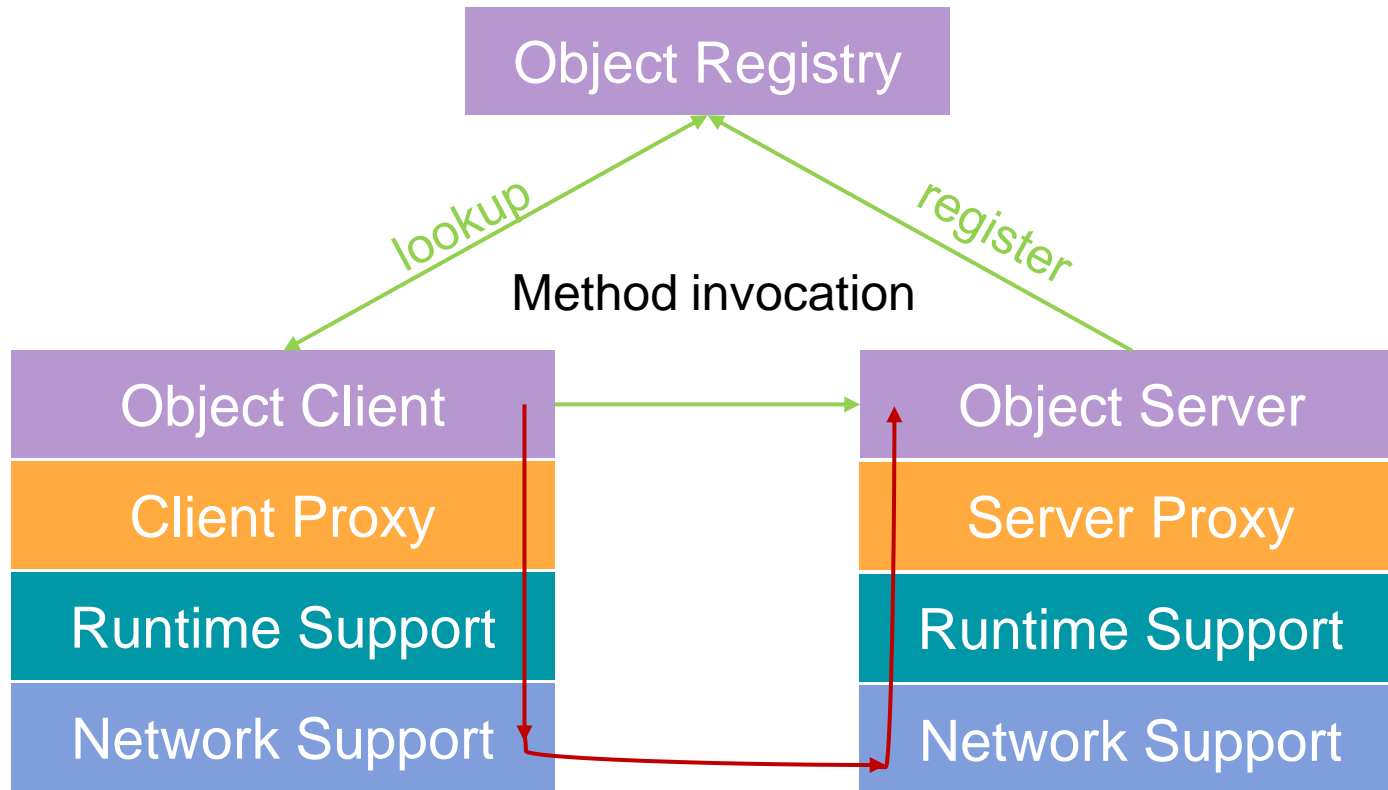
# Paradigma Distributed-Object

**Local Objects:** oggetti i cui metodi possono essere invocati solo da processi “locali”. Locale significa che il processo si trova sulla stessa macchina su cui si trova l’oggetto (*Object-Oriented Programming*)

**Distributed (Remote) Objects:** oggetti i cui metodi possono essere invocati da processi “remoti”. Remoto significa che il processo è in esecuzione su una macchina remota, diversa da quella su cui si trova l’oggetto interessato. Ovvio che deve esistere per forza una connessione (una rete) tra chi invoca il metodo e chi fornisce l’oggetto.

Le risorse disponibili in una rete sono rappresentate come **oggetti distribuiti**. I metodi relativi ad un oggetto distribuito sono detti **Remote Methods** (metodi remoti) mentre quelli relativi ad oggetti locali sono detti **Local Methods** (metodi locali).

# Architettura sistema Distributed-Object



→ Physical information flow

→ Logical information flow

Basata su architettura Client-Server.

Un processo *Object Server* rende disponibile un oggetto distribuito ai vari processi *Object Client*.

L'*Object Server* per rendere disponibile un oggetto distribuito deve effettuare un'operazione di *register* sull'*Object Registry*.

L'*Object Registry* mantiene un riferimento all'oggetto distribuito fornito dall'*Object Server* permettendone la localizzazione.

Un *Object Client*, per poter accedere ad un oggetto remoto, deve contattare (*lookup*) l'*Object Registry* per ottenere un object reference, che userà per invocare i metodi su questo oggetto.

# Distributed-Object System

## Lato Client

- Il Client invoca il metodo come se fosse un metodo locale.
- In realtà, la chiamata del metodo è gestita da un componente chiamato *client proxy* che interagisce con il software del client offerto dal *runtime support*.
- Il *runtime support* si prende cura delle chiamate inoltrate dal proxy all'oggetto remoto e gestisce il marshalling (il meccanismo di passaggio) dei parametri che devono essere trasmessi all'oggetto remoto.

## Lato Server

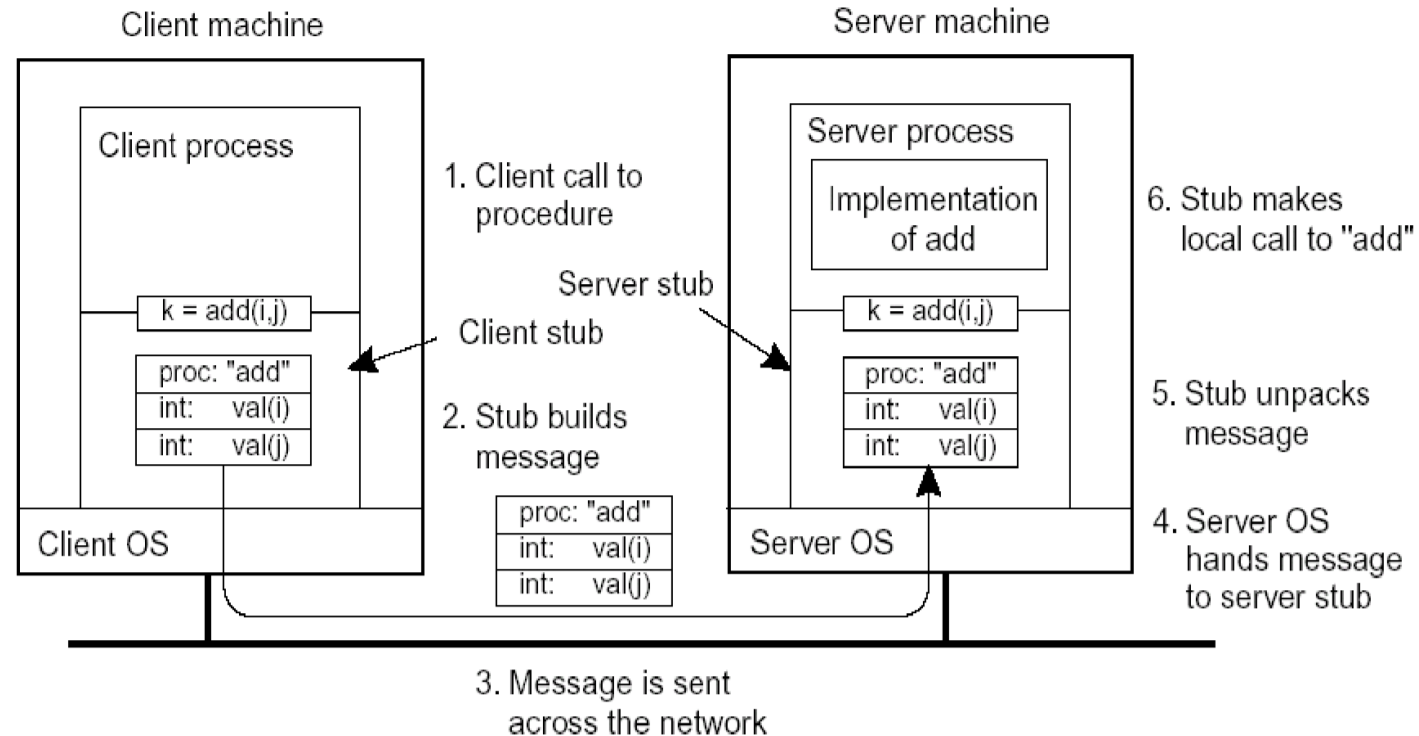
Attraverso il runtime support:

- gestisce i messaggi (richieste) che riceve
- effettua l'unmarshalling dei dati ricevuti
- inoltra la richiesta al *server proxy*
- il server proxy comunica con l'oggetto distribuito fornito dal server e invoca il metodo richiesto con i parametri ricevuti.

# Remote Procedure Call (RPC)

- Nel modello RPC una “chiamata a procedura” è inviata da un processo a un altro e i dati sono passati come argomenti.
- Quando il secondo processo riceve la chiamata, la procedura è eseguita localmente. Quando termina, si invia un messaggio che notifica il completamento al processo chiamante e il valore di ritorno (se previsto) viene inviato dal processo remoto al processo chiamante.
- L’RMI prende origine dall’RPC e altro non è che un’implementazione orientata agli oggetti del modello RPC ed è una API per i programmi Java.

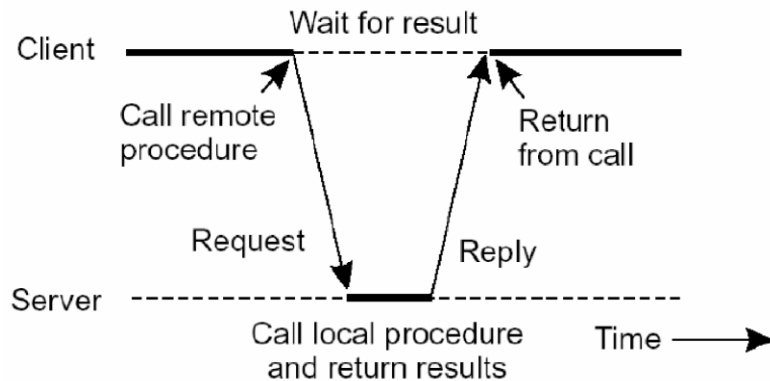
# Remote Procedure Call (RPC)



(1) La chiamata a procedura di un Client viene gestita da un *client stub*, (2) che la organizza in un messaggio specificando la procedura invocata e i parametri come dati. (3) Il messaggio viene spedito mediante la connessione di rete e ricevuto dal server. (4) A gestirlo vi è un server stub (5) che “spacchetta” il messaggio ricreando la chiamata alla procedura. (6) Lo stub effettua così la chiamata locale e parte l'esecuzione sul server.

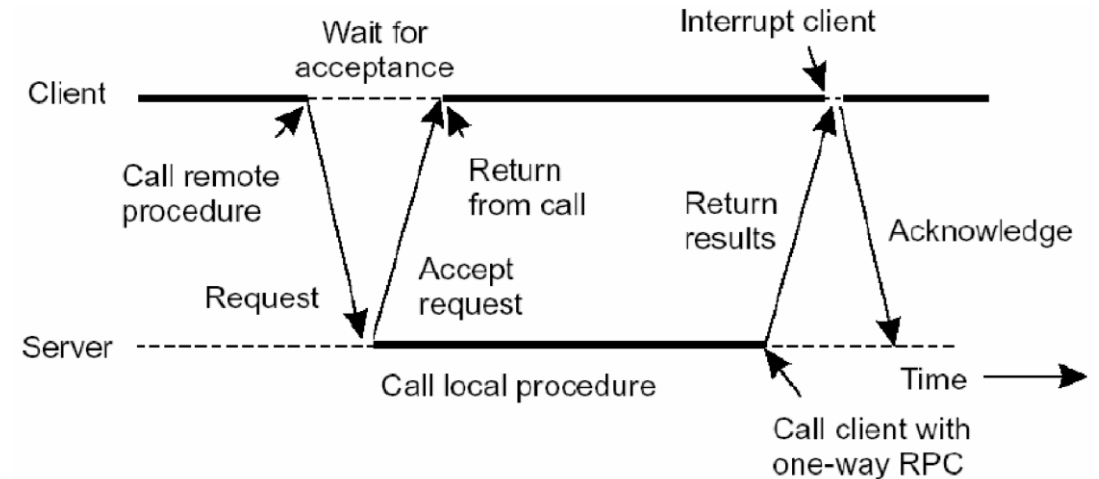
# Tipi di chiamata: sincrona e asincrona

## Synchronous Call



Dopo aver effettuato la chiamata, il client rimane in attesa del risultato senza fare altro per tutto il tempo necessario per le trasmissioni e per l'esecuzione del metodo sul server.

## Asynchronous Call



Il client invia la richiesta e aspetta subito un messaggio dal server che notifichi l'avvenuta ricezione della richiesta. Ricevuta la conferma, il client continua a fare altro mentre il server soddisfa la sua richiesta eseguendo il metodo remoto. Quando termina, il server invia il risultato con una one-way RPC al client. Questa effettua una interrupt sul client che riceve il risultato ed invia un ack al server.



# Java RMI

# Java RMI: principali componenti

L'obiettivo di Java RMI è quello di estendere il modello Java Object per supportare la programmazione con gli oggetti distribuiti e rendere la programmazione distribuita facile quanto quella standard.

- **Remote Objects:** sono normali oggetti Java ma la loro classe estende alcune classi RMI library che contengono il supporto per l'invocazione remota.
- **Remote references:** sono i riferimenti agli oggetti remoti contenuti su macchine remote.
- **Remote interfaces:** sono normali interfacce Java che specificano un oggetto remoto. Devono estendere l'interfaccia marker `java.rmi.Remote`. L'interfaccia remota deve essere nota sia al codice locale che al codice remoto

# Java RMI: tecnologie di supporto

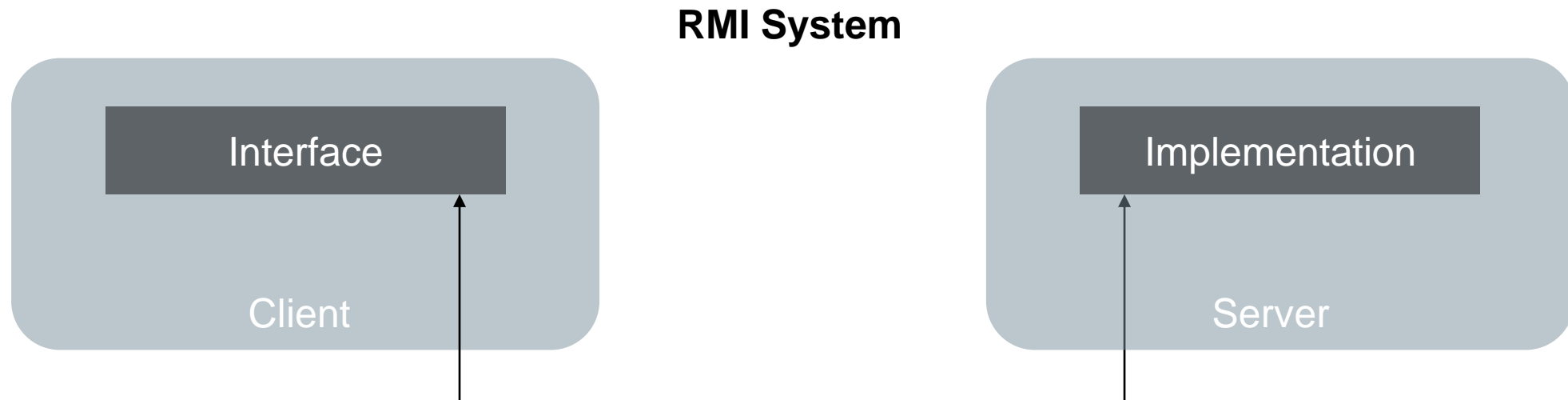
- **Registries:** i registri sono il posto in cui la macchina locale cerca i riferimenti per gli oggetti remoti.
- **Serialization:** il modello Java RMI fa necessariamente uso della rete. Per fare in modo che i processi remoti possano scambiarsi dati e risultati è necessario serializzare gli oggetti Java (byte stream). Necessaria per le operazioni di marshalling e unmarshalling dei parametri da passare ai metodi remoti.
- **Dynamic class loading:** il caricamento dinamico delle classi è necessario in diverse occasioni. Può succedere ad esempio che la chiamata di un metodo remoto sull'host A restituisca un oggetto di una classe non nota ad A ma solo alla macchina remota, l'host B.
- **Security Manager:** è il meccanismo usato per controllare il comportamento del codice caricato da un host remoto. Essendo un sistema distribuito la sicurezza ricopre un ruolo molto importante.

# Java RMI: struttura

Java RMI è basato sulle interfacce.

Tutti gli oggetti remoti devono implementare l'interfaccia `java.rmi.Remote`.

L'interfaccia definisce il comportamento e le classi definiscono l'implementazione.



In un sistema RMI l'interfaccia risiede sul client e l'implementazione è sul server, dove infatti il programma viene eseguito.

# RMI Registry

Prima che il Clien possa invocare un metodo su un oggetto remoto, è necessario che ottenga un reference (un riferimento) all'oggetto.

L'RMI Registry è un **Naming Service**, un servizio in esecuzione separatamente dal resto su una porta TCP (di default la 1099).

- I programmi server registrano presso l'RMI Registry gli oggetti remoti assegnando un nome a ogni oggetto.
- I programmi client ottengono dall'RMI Registry i riferimenti agli oggetti tramite il nome.

I nomi degli oggetti sul registry hanno un formato URL:

`rmi://hostname:port/remoteObjectName`

# Interfaccia RMI Registry

`void rebind (String name, Remote obj)` → this method is used by a server to register the identifier of a remote object by name

`void bind (String name, Remote obj)` → this method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference, an exception is thrown

`void unbind (String name, Remote obj)` → this method removes a binding

`Remote lookup(String name)` → this method is used by clients to look up a remote object by name. A remote object reference is returned.

LATO  
SERVER

LATO  
CLIENT

# Stubs e Skeletons

Uno **Stub** è una rappresentazione locale di un oggetto remoto: il client invoca i metodi dello Stub il cui compito sarà quello di invocare i metodi dell'oggetto remoto che rappresenta (spesso chiamato "proxy class").

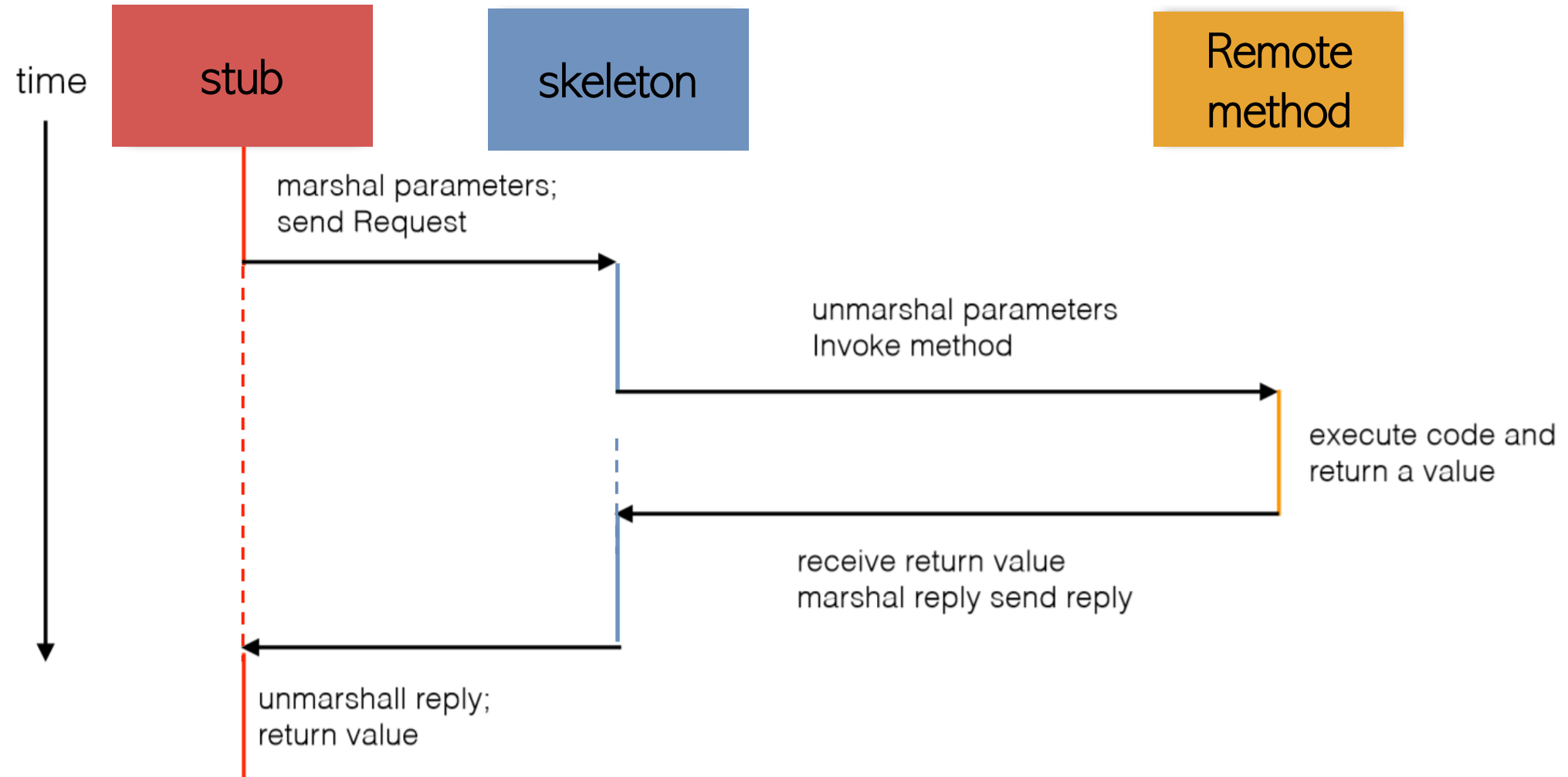
Quando viene invocato un metodo, lo Stub provvede a compiere le seguenti operazioni:

- Inizia una connessione con la Java Virtual Machine che contiene l'oggetto remoto, ovvero con lo Skeleton.
- Trasmette i parametri allo Skeleton (marshalling).
- Aspetta il risultato dell'invocazione del metodo.
- Legge il risultato o l'eccezione generata.
- Restituisce il risultato a chi ha invocato il metodo.

Ogni oggetto remoto ha sulla Java Virtual Machine un suo corrispondente **Skeleton** il cui scopo è quello di:

- Leggere i parametri di un'invocazione remota (unmarshalling).
- Invocare il metodo remoto dell'oggetto che rappresenta.
- Restituire il risultato al chiamante.

# Comunicazione tra Stubs e Skeletons





# Passaggio di parametri in RMI

Java standard:

- tipi primitivi: passaggio per valore
- oggetti: passaggio di riferimento

Java RMI:

- oggetti e tipi primitivi: passaggio per valore
- oggetti remoti: passaggio per riferimento

# Costruire un sistema Java RMI

Un sistema Java RMI prevede:

- interfaccia remota
- implementazione di un'interfaccia remota
- un server che ospiti gli oggetti remoti
- un servizio RMI per la pubblicazione degli oggetti remoti
- un programma client che usa il servizio offerto dal server

# Costruire un sistema Java RMI

## 1) Definire l'interfaccia remota

Un'interfaccia remota deve soddisfare i seguenti requisiti:

- Estendere l'interfaccia `java.rmi.Remote`.
- Dichiarare metodi che includano l'eccezione `java.rmi.RemoteException` nella clausola `throws`.

## 2) Implementare l'interfaccia remota

L'implementazione del comportamento di un oggetto avviene:

- Implementando l'interfaccia remota (quindi `java.rmi.Remote`).
- Estendendo la classe `UnicastRemoteObject`, il costruttore della classe ha nella clausola `throws` la `java.rmi.RemoteException` e chiama `super()` al suo interno.

# Costruire un sistema Java RMI

## 3) Creare il Server

- Si tratta di una Java Application che crea una o più istanze di oggetti remoti.
- Associa almeno uno degli oggetti remoti a un nome nel registro RMI con il metodo `Naming.rebind("...")`

## 4) Creare il Client

- Ottiene un riferimento remoto chiamando `Naming.lookup ("...")`
- Riceve un oggetto stub per l'oggetto remoto richiesto
- Richiama i metodi direttamente sul riferimento (così come accadrebbe per gli oggetti in Java standard)

# Esempio: Chat

Oggetto remoto: ChatServer

Operazioni ammesse:

- Login
- Logout
- chat per l'invio di messaggi

# Chat: Interfaccia Remota (1)

```
public interface ChatServer extends java.rmi.Remote {  
  
    public void login(String name, String password) throws  
        java.rmi.RemoteException;  
  
    public void logout(String name) throws  
        java.rmi.RemoteException;  
  
    public void chat(String name, String message) throws  
        java.rmi.RemoteException;  
  
}
```

# Chat: Implementazione Interfaccia Remota (2)

```
public class ChatServerImpl
extends java.rmi.server.UnicastRemoteObject implements ChatServer{

    public ChatServerImpl() throws java.rmi.RemoteException{
        super(); // serve per l'inizializzazione dell'oggetto remoto
    }

    public void login(String name, String pass) throws java.rmi.RemoteException{
        // Method Implementation
    }

    public void logout(String name) throws java.rmi.RemoteException{
        // Method Implementation
    }

    public void chat(String name, String msg) throws java.rmi.RemoteException{
        // Method Implementation
    }

}
```

# Chat: Object Server (3)

```
public class ChatServerAppl{

    public ChatServerAppl() {
        try {
            ChatServer c = new ChatServerImpl();
            Naming.rebind("rmi://localhost/ChatService", c);
        } catch (Exception e) {
            System.out.println("Server Error: " + e);
        }
    } //ChatServerApplconstructor

    public static void main(String args[]) throws java.rmi.RemoteException{
        //Create the new ChatServer
        new ChatServerAppl();
    } //main
} //ChatServerAppl
```



# Chat: Object Client (4)

```
public class ChatClient{
    public static void main(String[] args) throws Exception {
        try {
            // Get a reference to the remote object through the rmiregistry
            ChatServer c = (ChatServer) Naming.lookup("rmi://localhost/ChatService");
            // Now use the reference c to call remote methods
            c.login("Chas","*****");
            c.chat("Chas", "Hello");
            // Catch the exceptions that may occur -rubbish URL, Remote exception
        }
        catch (RemoteException re) {
            System.out.println("RemoteException"+re);
        }
    }
}
```

# Client-side Callback

- In molte architetture un server potrebbe aver bisogno di “chiamare” un client.
- Si fa uso delle **Client Callback**, un meccanismo che permette a un client di essere notificato da un server per un evento che il client aspettava.
- La notifica del server altro non è che l’invocazione di un metodo remoto sul Client.
- Per permettere l’uso di tale meccanismo il client RMI deve comportarsi come un server RMI.
- Lasciare che un Client estenda la classe `java.rmi.server.UnicastRemoteObject` è poco opportuno, quindi in questi casi, l’oggetto remoto (sul client) viene reso “pronto” per essere usato da remoto chiamando il metodo statico: `UnicastRemoteObject.exportObject(remote_object)`

# Client-side Callback: Lato Server

Sul server è necessario mantenere un riferimento al client sul quale deve essere effettuata la callback.

1. Definire l'interfaccia remota del server (normalmente)
2. aggiungere tra i metodi remoti quello che il client invocherà per inviare il riferimento sul quale verrà fatta la callback
3. Implementare l'interfaccia remota sul server (normalmente)
4. Creare un'istanza dell'implementazione dell'interfaccia remota del server e pubblicarla sul registro (normalmente).

# Client-side Callback: Lato Client

Sul client è necessario fornire il metodo remoto che il server invocherà per eseguire la callback.

Si seguono 3 passaggi:

1. Definire l'interfaccia remota sul client nella quale si dichiara il metodo invocato dal server per eseguire la callback (ovviamente questa interfaccia estenderà `java.rmi.Remote`).
2. Implementare l'interfaccia remota del client (cioè implementare il metodo di callback).
3. Creare un'istanza dell'implementazione dell'interfaccia remota e passarla al server che la userà per eseguire la callback.

# Esempio: Timer

- Un Timer fornisce la data e l'ora corrente
- TimeClient desidera ricevere aggiornamenti periodici (call-backs) con la data e l'ora correnti (fungerà da server in ascolto degli aggiornamenti).
- Sul Server
  1. Definire l'interfaccia TimeServer del server (normalmente) e aggiungere il metodo "registerTimeMonitor" (utilizzato dal client per inviare il riferimento su cui deve essere effettuato il callback)
  2. Implementare il "TimeServerImpl" oggetto remoto, creare un'istanza e pubblicarla sul registro (normalmente)
- Sul Client
  1. Definire l'interfaccia TimeMonitor per il client, incluso il metodo ("time") utilizzato per il call-back (che sarà invocato dal server)
  2. Implementare "TimeClient" (implementando l'interfaccia "TimeMonitor"), che deve registrarsi sul server invocando il metodo "registerTimeMonitor" (passando come parametro un riferimento a se stesso)

# Timer

```
import java.rmi.*;  
import java.util.Date;
```

```
public interface TimeMonitor extends java.rmi.Remote {  
    public void time(Date d) throws RemoteException;  
}
```

← Sarà invocato dal server  
per la callback del client

-----

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

```
public interface TimeServer extends java.rmi.Remote {  
    public void registerTimeMonitor(TimeMonitor tm)  
    throws RemoteException;  
}
```

← Sarà invocato dal client  
per inviare il riferimento  
sul quale deve essere  
effettuata la call-back

# Timer: TimeTicker sul Server

```
import java.util.Date;
class TimeTicker extends Thread {
    private TimeMonitor tm;

    TimeTicker( TimeMonitor tm ){
        this.tm = tm;
    }

    public void run() {
        while(true)
            try{
                tm.time(new Date());
                sleep( 2000 );
            } catch ( Exception e ) { System.out.println(e); }
    } //run
} //TimeTicker clas
```

# Timer: Server

```
import java.net.*;
import java.io.*;
import java.util.Date;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;

public class TimeServerImpl extends UnicastRemoteObject implements
TimeServer {

    public void registerTimeMonitor( TimeMonitor tm ) {
        System.out.println( "Client requesting a connection" );
        TimeTicker tt; tt = new TimeTicker( tm );
        tt.start();
        System.out.println( "Timer Started" );
    } ...
}
```



# Timer: Server

...

```
private static TimeServerImpl tsi;

public static void main (String[] args) {
    try {
        tsi = new TimeServerImpl();
        LocateRegistry.createRegistry(1099);
        System.out.println("Registry created");
        Naming.rebind("TimeServer", tsi);
        System.out.println( "Bindings Finished" );
        System.out.println( "Waiting for Client requests" );
    } catch (Exception e) { System.out.println(e); }
} //main

} // class TimeServerImpl
```

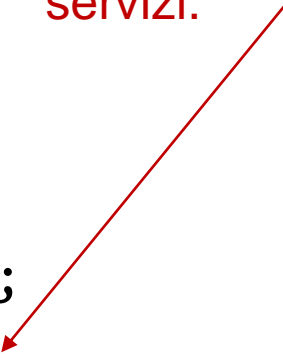
# Timer: Client

```
import java.util.Date;
import java.net.URL;
import java.rmi.*;
import java.rmi.server.*;

public class TimeClient implements TimeMonitor {
    private TimeServer ts;

    public TimeClient() {
        try {
            System.out.println( "Exporting the Client" );
            UnicastRemoteObject.exportObject(this,1098);
            ts = (TimeServer)Naming.lookup(
                "rmi://localhost:1099/TimeServer");
            ts.registerTimeMonitor(this);
        } catch (Exception e){ System.out.println(e); }
    } //constructor ...
}
```

Rende il client pronto per l'uso "remoto". Ricorda di usare una porta che non è in uso da altri servizi.



# Timer: Client

...

```
public void time( Date d ){  
    System.out.println(d);  
}
```

```
public static void main (String[] args) {  
    new TimeClient();  
}
```

```
} //class TimeClient
```

# Sicurezza

Un'applicazione Java RMI comporta il download di codice (stub downloading) sulla macchina client per avere lo stub dell'oggetto remoto.

Il download di codice è gestito e controllato da un Security Manager che agisce nel rispetto di una policy specifica. Controlla che tutte le operazioni contenute nel codice dello stub rispettino le regole stabilite nel file di policy.

Prima che un'applicazione Java abbia il permesso di scaricare codice dinamicamente, è necessario che siano settati un **security policy** (un file di testo) e un **security manager** adatto.

# Security Policy

Il security policy è specificato in un file di testo.

Il file policy usato in RMI specifica che il codice scaricato da un **codebase** è permesso. Un codebase è un insieme di classi cui si vogliono assegnare permessi.

Es.

```
grant {  
    permission java.net.SocketPermission "*:1024-65535","connect,accept";  
    permission java.net.SocketPermission "*:80", "connect";  
    // permission java.security.AllPermission; // be careful!!!  
};
```

Si può avere un blocco grant per ogni codebase cui si vogliono assegnare permessi diversi.

# Settare Security Policy e Security Manager

## **Come settare un file di policy**

Il file di policy può essere settato da linea di comando in fase di lancio dell'applicazione settando `java -Djava.security.policy=nome_file_policy`, oppure utilizzando `System.setProperty()` direttamente nel codice del programma.

## **Come settare il Security Manager**

Se non si setta un security manager, il file policy viene ignorato. Quindi è fondamentale settarne uno. Esistono 2 modi per dichiarare un Security Manager:

- da linea di comando, settando `java -Djava.security.manager;`
- dichiarandolo esplicitamente nel codice usando `System.setSecurityManager(new RMISecurityManager())`. Questo deve essere fatto in tutte le applicazioni che devono scaricare codice.

# Dynamic Class Downloading

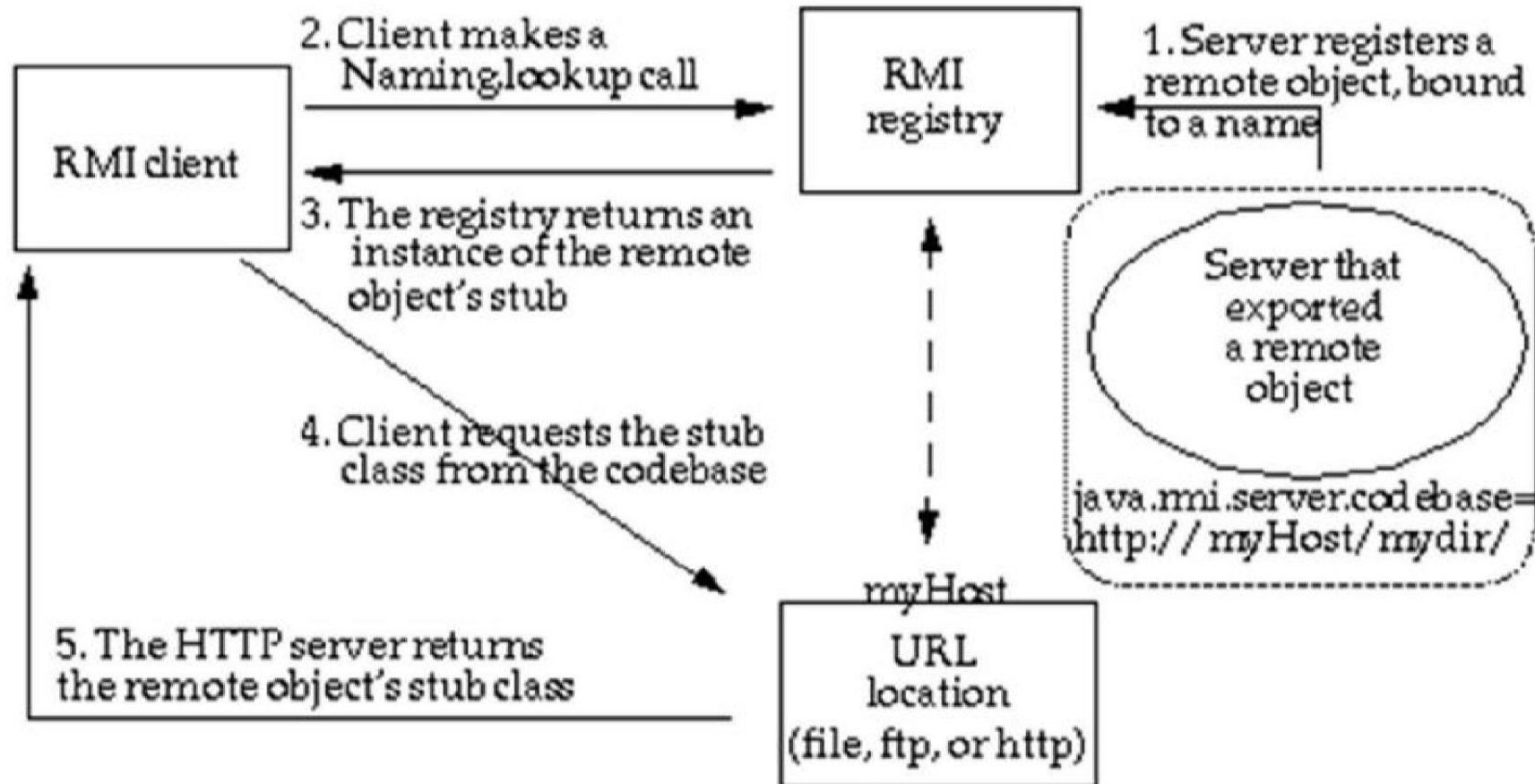
- Un'importante caratteristica della piattaforma Java è la possibilità di scaricare dinamicamente ed eseguire classi da un URL (Uniform Resource Locator). In questo modo, una JVM può eseguire un'applicazione che non è stata definita sul sistema su cui è in esecuzione (recupera classi remote).
- Quando un'applicazione RMI viene eseguita, i file contenenti le classi di supporto all'applicazione devono risiedere in una locazione accessibile sia dal server che dal client.
- Una volta noti i file che devono essere sui diversi nodi della rete, è semplice assicurare che siano disponibili per il Class Loader di ogni JVM.

# Dynamic Class Downloading: concetti chiave

- I progettisti RMI hanno esteso il concetto di class loading includendo il caricamento di classi da server FTP e HTTP.
- Solitamente, un class loader è accoppiato ad un server HTTP che rende disponibili le classi.
- Il codebase è definito in modo tale da specificare dove si trovano le classi di cui ha bisogno la JVM per eseguire un programma.
- Per poter supportare il class loading remoto, in Java RMI si fa uso dell'*RMIClassLoader*.



# Dynamic Class Downloading: use codebase



# Dynamic Class Downloading: uso codebase

1. Il codebase dell'oggetto remoto è specificato dal server attraverso la proprietà *java.rmi.server.codebase*:  
*java -Djava.rmi.server.codebase=http://webserver/export/*  
Il server RMI registra l'oggetto sul RMI registry. Il codebase impostato sul server JVM è collegato al riferimento dell'oggetto remoto nell'RMI registry.
2. Quando il client ha bisogno dell'oggetto remoto, effettua la lookup sul RMI registry.
3. L'RMI registry restituisce al client un'istanza dello stub dell'oggetto remoto su cui può operare il client.

# Dynamic Class Downloading: uso codebase

4. Per poter usare l'oggetto, il client ha bisogno delle classi ad esso relative. Se le classi vengono trovate nel CLASSPATH locale del client, vengono caricate. Se invece non vengono trovate, il client le cerca nel codebase dell'oggetto remoto.  
Il codebase sfruttato dal client è l'URL ricevuto insieme all'istanza dello stub dell'oggetto remoto ricevuta. Effettua così la richiesta delle classi stub direttamente sulla locazione indicata dall'URL del codebase.
5. Il server http (o ftp) restituisce le classi stub dell'oggetto remoto e il client può così caricarle e lavorare sull'oggetto remoto invocando i metodi su esso.

# Distributed Garbage Collector

In Java il programmatore non deve occuparsi di rilasciare la memoria allocata, il Garbage Collector (GC) libera la memoria da oggetti non più utilizzati.

RMI fornisce un ***Distributed Garbage Collector (DGC)*** basato sulla tecnica *reference counting* (conteggio dei riferimenti).

Meccanismo DGC:

Il Server mantiene il reference counting, mentre dal Client, in particolare dallo stub partono due messaggi, il primo quando è stabilito il riferimento e il secondo quando quest'ultimo viene meno.

Il DGC è esso stesso un server RMI che implementa un'interfaccia remota DGC definita nel package `java.rmi.dgc`.

# Distributed Garbage Collector

Questa interfaccia offre due metodi:

```
public void clean(ObjID[] ids, long seqNum, ....)
```

```
public Lease dirty(ObjID[] ids, long seqNum, Lease lease)
```

- `dirty()` è invocato dal Client quando richiede il riferimento all'oggetto remoto, il server incrementa il contatore
- `clean()` è invocato dal Client quando non utilizza più il riferimento all'oggetto remoto, il server decrementa il contatore.
- quando il contatore giunge a zero, l'oggetto remoto è libero da qualsiasi client; viene messo nella *weak reference list* ed è soggetto a periodiche "passate" del garbage collector.