Laboratorio di Reti – A (matricole pari)

Autunno 2021,

instructor: Laura Ricci

laura.ricci@unipi.it

Lezione 11

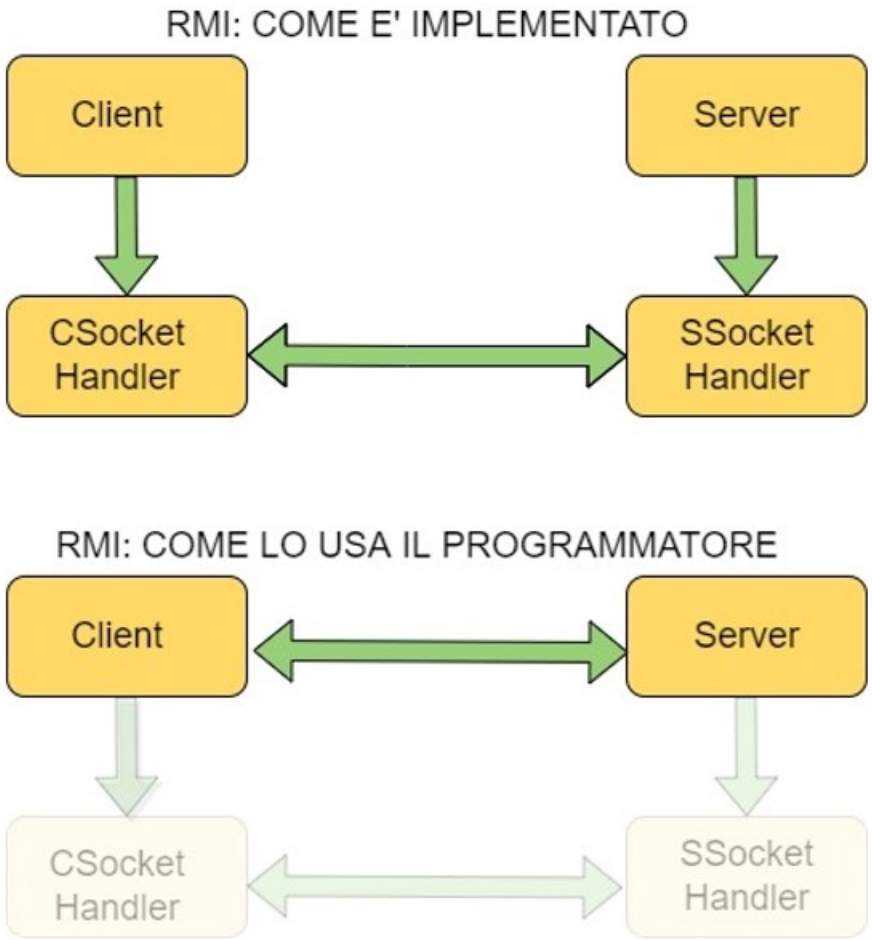
REMOTE METHOD INVOCATION CALLBACK RMI

30/11/2021

RMI: OLTRE I SOCKETS....

- sockets e RMI, due meccanismi che svolgono la stessa funzione ovvero permettere la comunicazione tra un client ed un server
  - ma in due modi diversi
- RMI
  - fornisce un livello di astrazione maggiore, rendendo trasparenti al programmatore l'uso dei socket TCP
    - l'implementazione è basata comunque su TCP
  - evita la gestione esplicita di thread lato server
  - Interoperabilità limitata : sia il client che il server devono essere scritti in JAVA

RMI IN BREVE

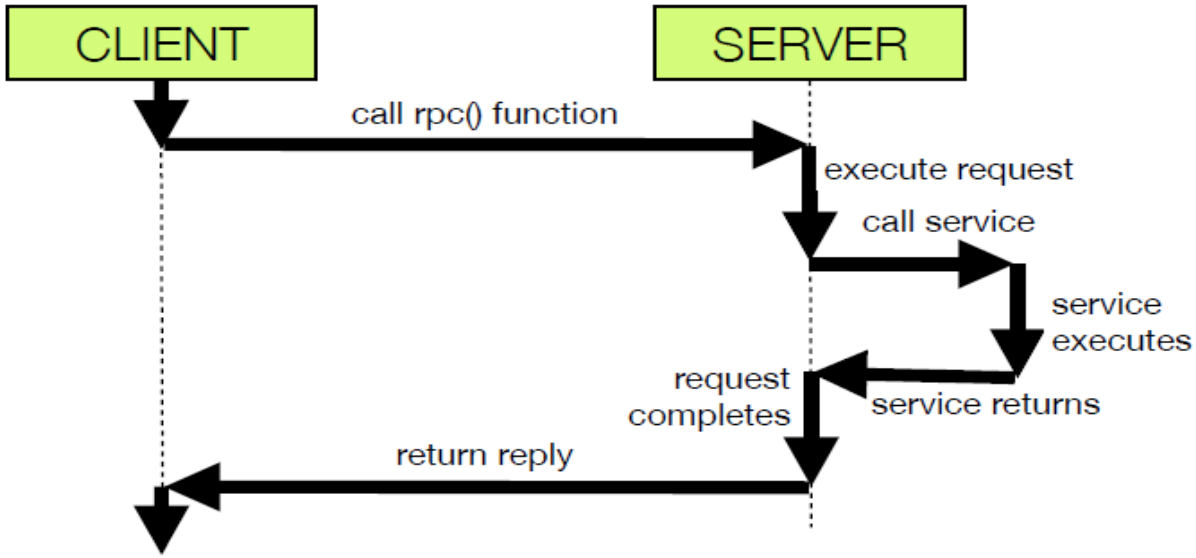

RMI: COME E' IMPLEMENTATO
Client
Server
CSocket Handler
SSocket Handler
RMI: COME LO USA IL PROGRAMMATORE
Client
Server
CSocket Handler
SSocket Handler

I PRECEDENTI: REMOTE PROCEDURE CALL

- esempio: un client richiede ad un server la stampa di un messaggio.
  - il server restituisce un codice che indica l'esito della operazione
  - il client attende l'esito dell'operazione
- la richiesta al server è implementata come l' invocazione di una procedura definita sul server
- i meccanismi utilizzati dal client sono gli stessi utilizzati per una normale invocazione di procedura, ma …
  - l'invocazione di procedura avviene sull'host su cui è in esecuzione il client
  - la procedura viene eseguita sull' host su cui è in esecuzione il server
  - i parametri della procedura vengono inviati automaticamente sulla rete dal supporto all'RPC

REMOTE PROCEDURE CALL (RPC)

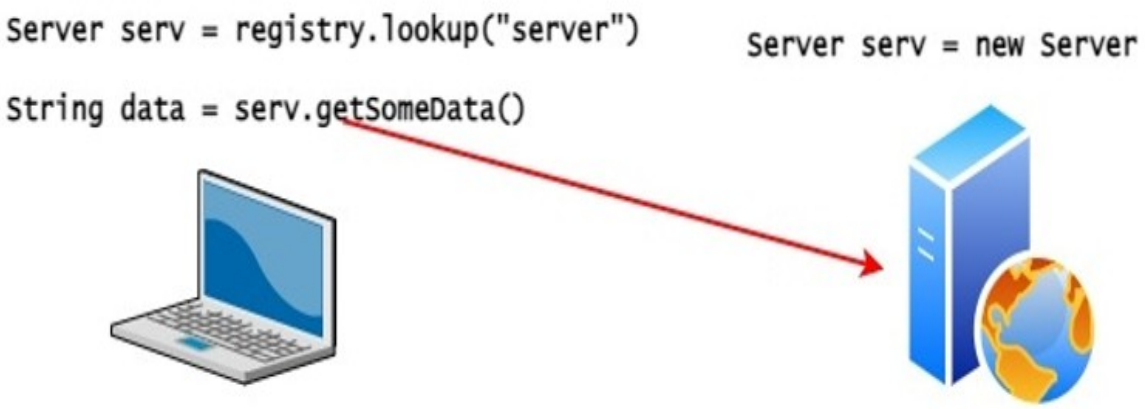
- paradigma di interazione a domanda/risposta
  - il client invoca una procedura del server remoto
  - il server esegue la procedura con i parametri passati dal client e restituisce a quest'ultimo il risultato dell'esecuzione.
  - la connessione remota è trasparente rispetto a client e server
- in genere prevede meccanismi di affidabilità a livello sottostante



REMOTE PROCEDURE CALL: LIMITI

- parametri e risultati devono avere tipi primitivi
- la programmazione è essenzialmente procedurale
  - non è basata sulla programmazione ad oggetti, quindi mancano i concetti di ereditarietà, incapsulamento, polimorfismo, …
- la localizzazione del server non è trasparente
  - il client deve conoscere l'IP e la porta del servizio
- il paradigma RPC è quindi evoluto verso il paradigma ad oggetti distribuiti
- In JAVA RMI: un paradigma ad alto livello per costruire applicazioni distribuite
  - che coinvolgono diverse JVM
  - possibilmente in esecuzione su host differenti comunicanti (un'applicazione multithreaded non è distribuita !)

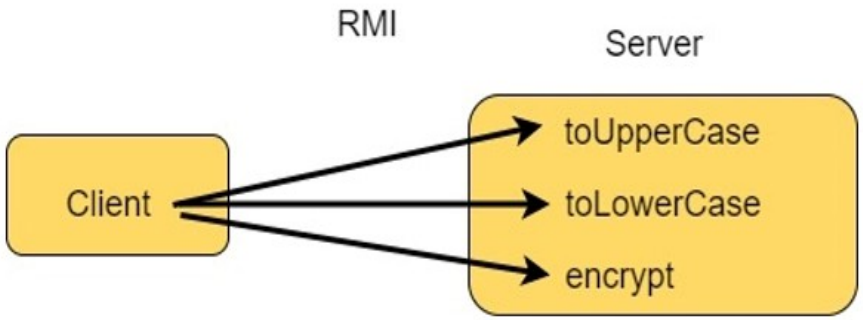
RMI IN BREVE



- il server crea l'oggetto remoto che fornisce il servizio
- il client invoca direttamente i metodi dell'oggetto remoto
  - `registry.lookup`: rappresenta un set up in cui si stabilisce una connessione con il server
  - l'interazione consiste semplicemente nell'invocazione di un metodo

RMI E SOCKET A CONFRONTO

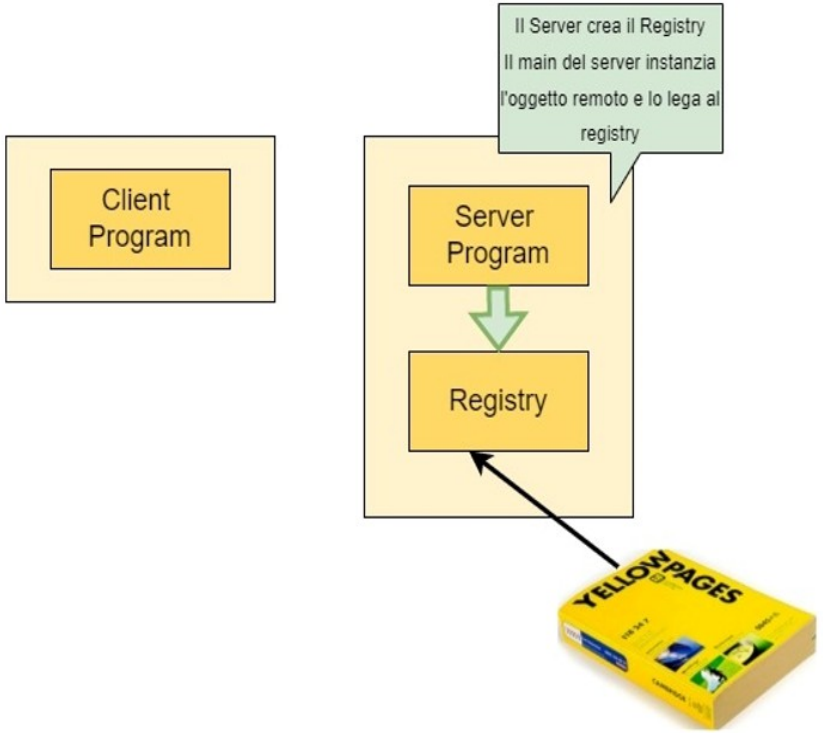

```
public void run()}
 try {
  Request request = (Request)
            inFromClient.readObject();
  if ("toUpperCase".equals(request.getType())
       {outToClient.writeObject(new
       Request("Uppercase",result));}
  else if ("toLowerCase".equals(request.getType())
       {outToClient.writeObject(new
       Request("LowerCasecase",result));}
  else if ("Encrypt".equals(request.getType())
       {outToClient.writeObject(new
       Request("Encrypt",result));
    }
  } catch(Exception e) {e.printStackTrace();}
}
```



- il client invoca metodi del server
- non un grosso switch nel codice del server (vedi codice a sinistra)
  - evita gestione "low level" delle invocazioni di metodo
- multithreading gestito dal server
  - ma sincronizzazione tra thread a carico del programmatore!

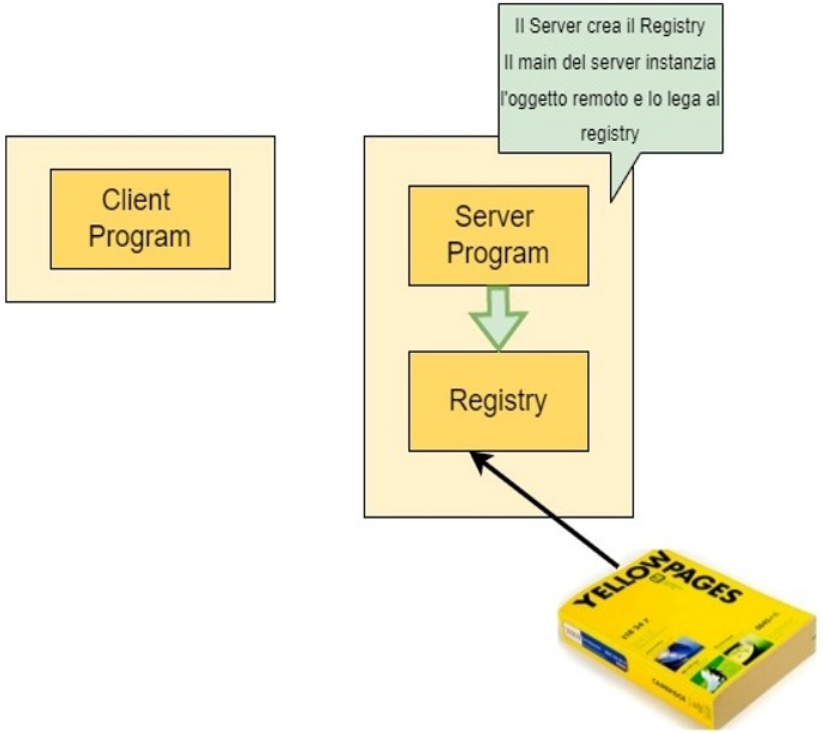
RMI SET UP

- tre entità fondamentali: client, server e registry
- a cosa serve il registry?
  - un servizio che agisce da 'yellow pages'
  - una applicazione separata dal server
  - in generale in esecuzione sull'host del server, ma anche su un host diverso
  - permette il bootstrap: mette in comunicazione client e server



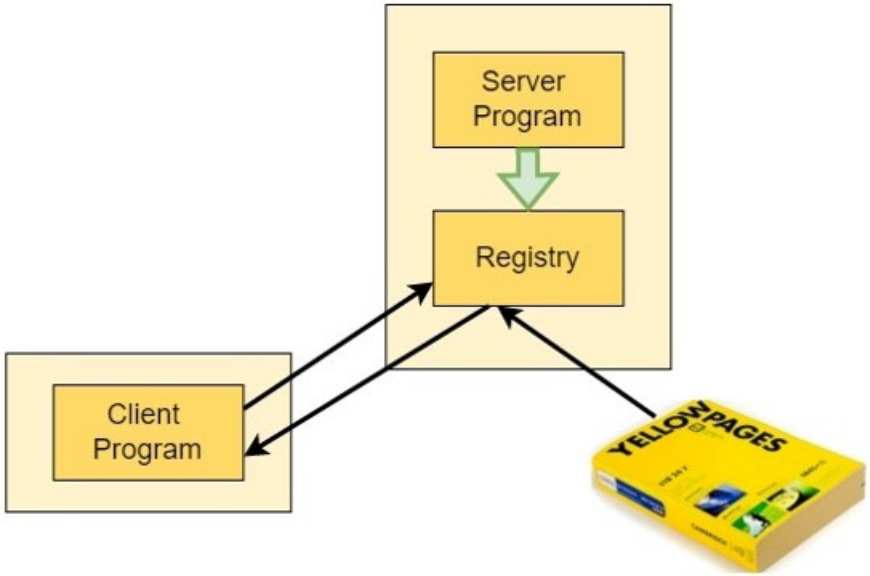
RMI SET UP: SERVER

- il server
  - istanzia l'oggetto remoto
  - lo esporta creando uno `stub` dell'oggetto
  - inserisce un riferimento a quell'oggetto nel `Registry`
  - bind: operazione utilizzata per la registrazione



RMI SET UP: LATO CLIENT

- il client
  - deve individuare un riferimento all'oggetto remoto nel registry
  - lookup: ricerca il riferimento all'oggetto remoto nel `Registry`,
  - riceve dal registry l'istanza di uno `stub` dell'oggetto remoti
- cosa è lo `Stub`?
  - "rappresentazione locale" di un oggetto remoto presso il client

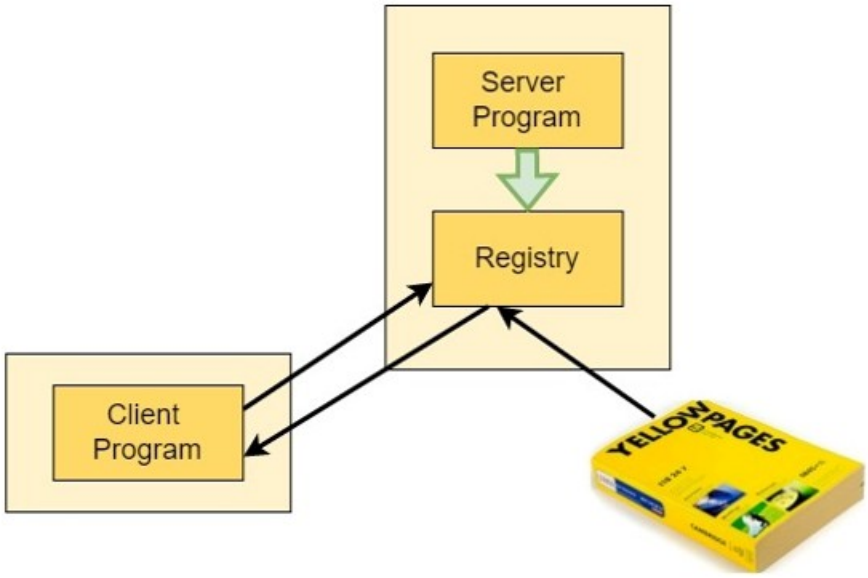


RMI SET UP: LATO CLIENT

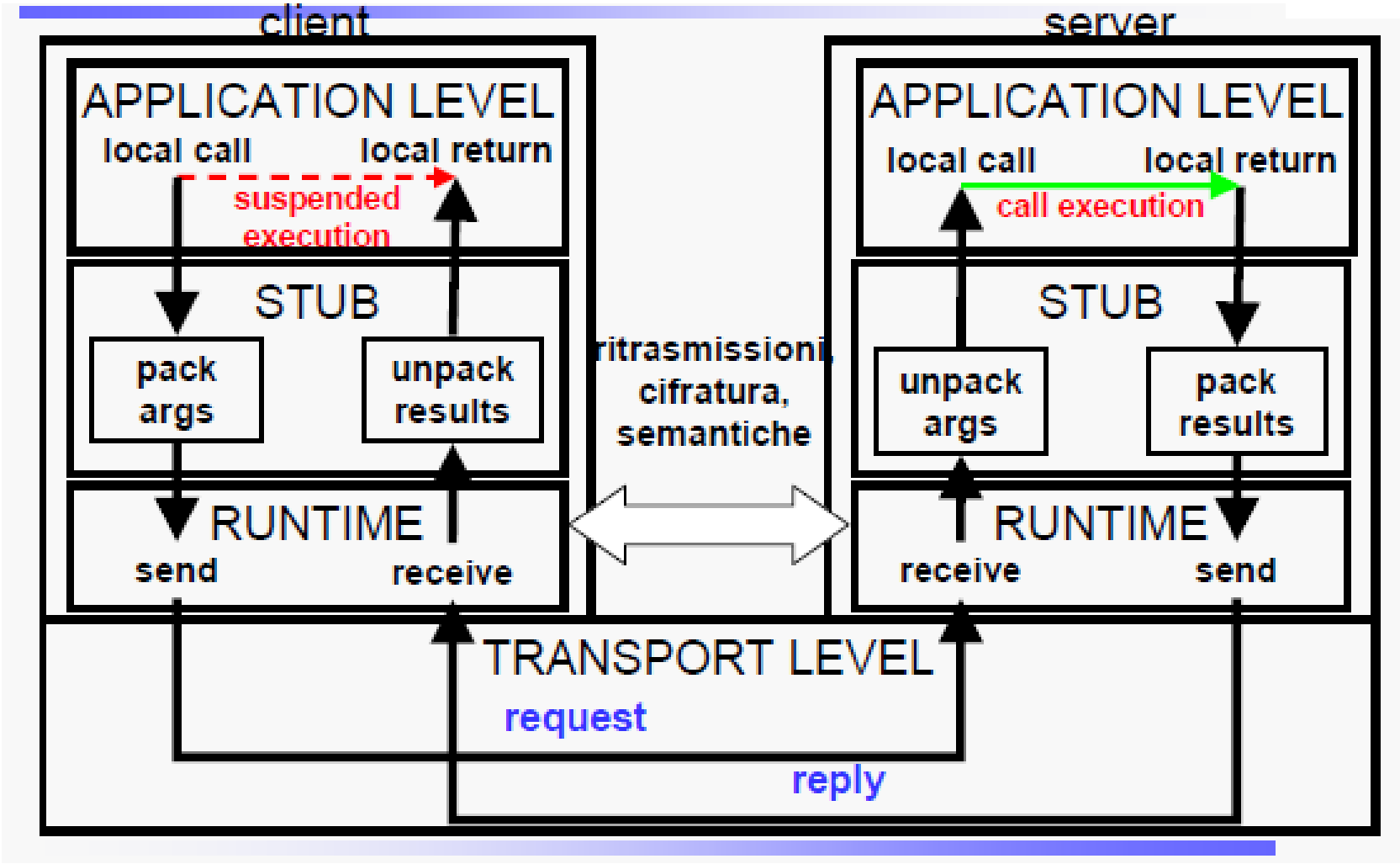
- il client invoca i metodi remoti come se fossero locali

```
OggettoOvunque cc;
cc.metodo  ()
```

- a livello logico: invocazione identica a quella di un metodo locale
- a livello di supporto:
  - trasformazione dei parametri della chiamata remota in dati da spedire sulla rete.
  - invio vero e proprio dei dati sulla rete



RMI "UNDER THE HOOD"



JAVA RMI: GENERALITA'

- un oggetto remoto è un oggetto i cui metodi possono essere acceduti da un diverso spazio di indirizzamento
  - una JVM diversa, potenzialmente in esecuzione su un altro host
- sfrutta le caratteristiche della programmazione ad oggetti
- tutte le funzionalità standard di `JAVA` sono disponibili in `JAVA RMI`

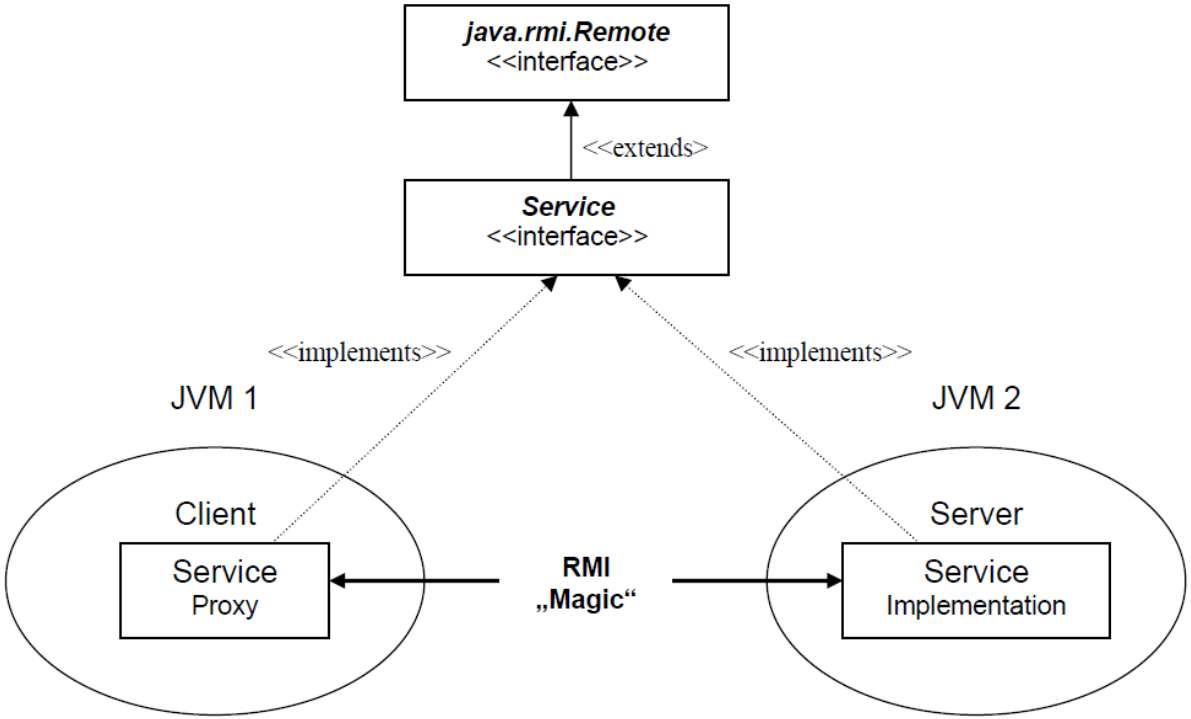
  meccanismi di sicurezza, serializzazione dei dati, ….
- supporta un `Java Security Manager` per controllare che le applicazioni distribuite abbiano i diritti necessari per essere eseguite.
- supporta un meccanismo di `Distributed Garbage Collection (DGB)` per disallocare quegli oggetti remoti per cui non esistano più referenze attive.

RMI STEP BY STEP

- introduciamo il framework RMI mediante un esempio, step by step
- definire un server che implementi, mediante RMI, il seguente servizio:
  - su richiesta del client, il server restituisce le principali informazioni relative ad un paese dell'Unione Europea di cui il client ha specificato il nome
    - linguaggio ufficiale
    - popolazione
    - nome della capitale

RMI : L'INTERFACCIA REMOTA

- interfaccia JAVA che dichiara i metodi accessibili da remoto
- è implementata da due classi:
  - sul server, la classe che implementa il servizio
  - sul client, la classe che implementa il proxy del servizio remoto



STEP 1: DEFINIZIONE DELL'INTERFACCIA REMOTA

- i metodi dell'oggetto remoto devono essere definiti in un'interfaccia remota
  - estende `java.rmi.Remote` o un'altra interfaccia che estenda `java.rmi.Remote`
  - è una (tag interface): non definisce alcun metodo, il solo scopo è solo quello di identificare gli oggetti che possono essere utilizzati in remoto

- i metodi remoti devono dichiarare di sollevare `Remote Exception`, della classe `java.rmi.RemoteException`

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface IntRemota extends Remote {
        public int remoteHash (String s) throws RemoteException;}
```

- se non si solleva `Remote Exception`, nella segnatura del metodo, si ottiene l' eccezione `InvalidInterface`

STEP 1: DEFINIZIONE DELL'INTERFACCIA REMOTA

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EUStatsService extends Remote {
    String getMainLanguages(String CountryName)
                            throws RemoteException;

    int getPopulation(String CountryName)
                            throws RemoteException

    String getCapitalName(String CountryName)
                            throws RemoteException;
                                                }
```

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

definire una classe che implementa i metodi della interfaccia remota

```
public class MyServerRemoto extends MyClass implement IntRemota
     { public MyServerRemoto() throws RemoteException {.....}
        ..... }
```

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

- utilizziamo una classe di appoggio e una hash table per memorizzare i dati riguardanti le nazioni europee.
- in questo modo definiamo un semplice data base delle nazioni
- classe `EUData` per definire oggetti (record) che descrivono la singola nazione

```
class EUData {
    private String Language;
    private int population;
    private String Capital;
    EUData(String Lang, int pop, String Cap) {
            Language = Lang;
            population = pop;
            Capital = Cap;
            }
   String getLangs( ) { return Language; }
   int getPop( )      { return population; }
   String getCapital( ) { return Capital; } }
```

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

```
import java.rmi.*;              // Classes and support for RMI
import java.rmi.server.*;          // Classes and support for RMI servers
import java.util.Hashtable;    // Contains Hashtable class
public class EUStatsServiceImpl implements EUStatsService {
   /* Store data in a hashtable */
   Hashtable <String, EUData> EUDbase = new Hashtable<String, EUData>();
   /* Constructor - set up database */

    EUStatsServiceImpl() throws RemoteException {
       EUDbase.put("France", new EUData("French",57800000,"Paris"));
       EUDbase.put("United Kingdom", new EUData("English",
                                                    57998000,"London"));
       EUDbase.put("Greece", new EUData("Greek",10270000,"Athens"));
       ........  }
```

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

```
/* implementazione dei metodi dell'interfaccia */
public String getMainLanguages(String CountryName)
                                              throws RemoteException {
            EUData Data = (EUData) EUDbase.get(CountryName);
            return Data.getLangs();}
public int getPopulation(String CountryName)
                                            throws RemoteException {
            EUData Data = (EUData) EUDbase.get(CountryName);
            return Data.getPop();    }
public String getCapitalName(String CountryName)
                                            throws RemoteException {
            EUData Data = (EUData) EUDbase.get(CountryName);
            return Data.getCapital( ); }
```

STEP 3: ESPORTAZIONE DEL SERVIZIO

- esportare l'oggetto
- diversi modi per esportare l'oggetto, per il momento utilizzeremo

```
UnicastRemoteObject.exportObject(remoteobj, 0);
```

- il metodo statico `UnicastRemoteObject.exportObject` esporta l'oggetto, ovvero
  - una socket su una porta anonima per accettare le richieste TCP
  - restituisce lo `stub` dell'oggetto remoto da passare al client

PASSO 3: GENERAZIONE STUB

- `Stub`
  - è un oggetto che consente di interfacciarsi con un altro oggetto (il target) in modo da sostituirsi ad esso
  - si comporta da intermediario: inoltra le chiamate che riceve al suo target
  - contiene le informazioni (indirizzo IP e porta) per contattare l'oggetto remoto

- per generare un'istanza dello `Stub`: generare stato oggetto + metodi dell'oggetto

- meccanismi diversi nelle diverse versioni di JAVA
  - `RMI compiler`, nelle prime versioni <1.5
  - `Reflection`, nelle versioni più recenti: noi utilizeremo questo meccanismo

STEP 3: ESPORTAZIONE DEL SERVIZIO

- definiamo il main della classe `EuStatServer` che crea implementa la interfaccia `EUStats`
- contiene il "launch code" dell'oggetto remoto
  - crea una istanza dell'oggetto remoto
  - esporta l'oggetto remoto
  - associa all'oggetto remoto un nome simbolico e registra lo stub nel registry

PASSO 3: ATTIVAZIONE E GENERAZIONE STUB

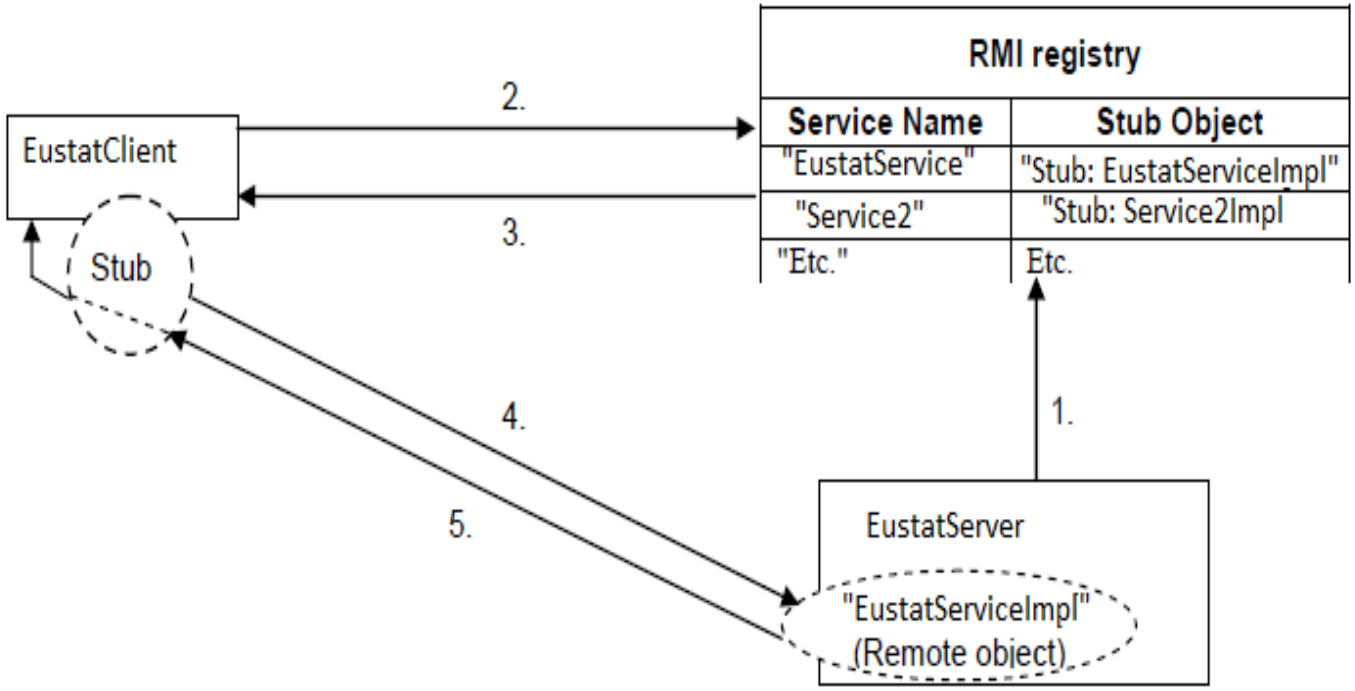
```
public static void main (String args[]) {
try {
    /* Creazione di un'istanza dell'oggetto EUStatsService */
    EUStatsServiceImpl statsService = new EUStatsServiceImpl();
    /* Esportazione dell'Oggetto */
    EuStats stub = (EuStats)
          UnicastRemoteObject.exportObject(statsService, 0);
   /* Creazione di un registry sulla porta args[0]
   LocateRegistry.createRegistry(args[0]);
   Registry r=LocateRegistry.getRegistry(args[0]);
   /* Pubblicazione dello stub nel registry */
   r.rebind("EUSTATS-SERVER", stub);
   System.out.println("Server ready");}
/* If any communication failures occur... */
catch (RemoteException e) {
System.out.println("Communication error " + e.toString());}}}
```

PASSO 3: ATTIVAZIONE E GENERAZIONE STUB

Il main della classe `EustatServer`:

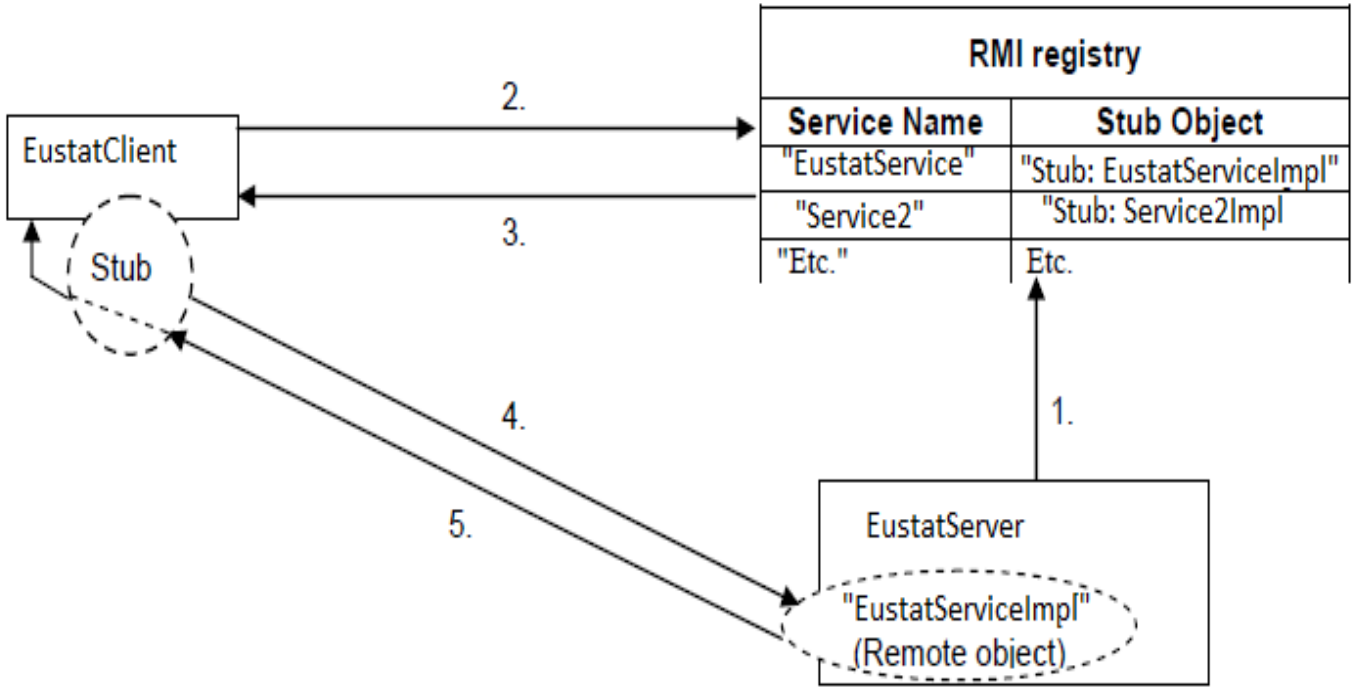
- crea un'istanza del servizio (oggetto remoto)
- invoca il metodo statico `UnicastRemoteObject.exportObject(obj,0)` che esporta dinamicamente l'oggetto,
  - indicando la porta 0 viene utilizzata una porta anonima scelta dal supporto
- restituisce un'istanza dell'oggetto, che "rappresenta" l'oggetto remoto mediante il suo riferimento
- pubblica il riferimento all'oggetto remoto nel registry
- il main quindi termina, ma
  - il thread in attesa di invocazione di metodi remoti rimane attivo
  - il server non termina

JAVA RMI > 1.5: THE EUSTAT APPLICATION



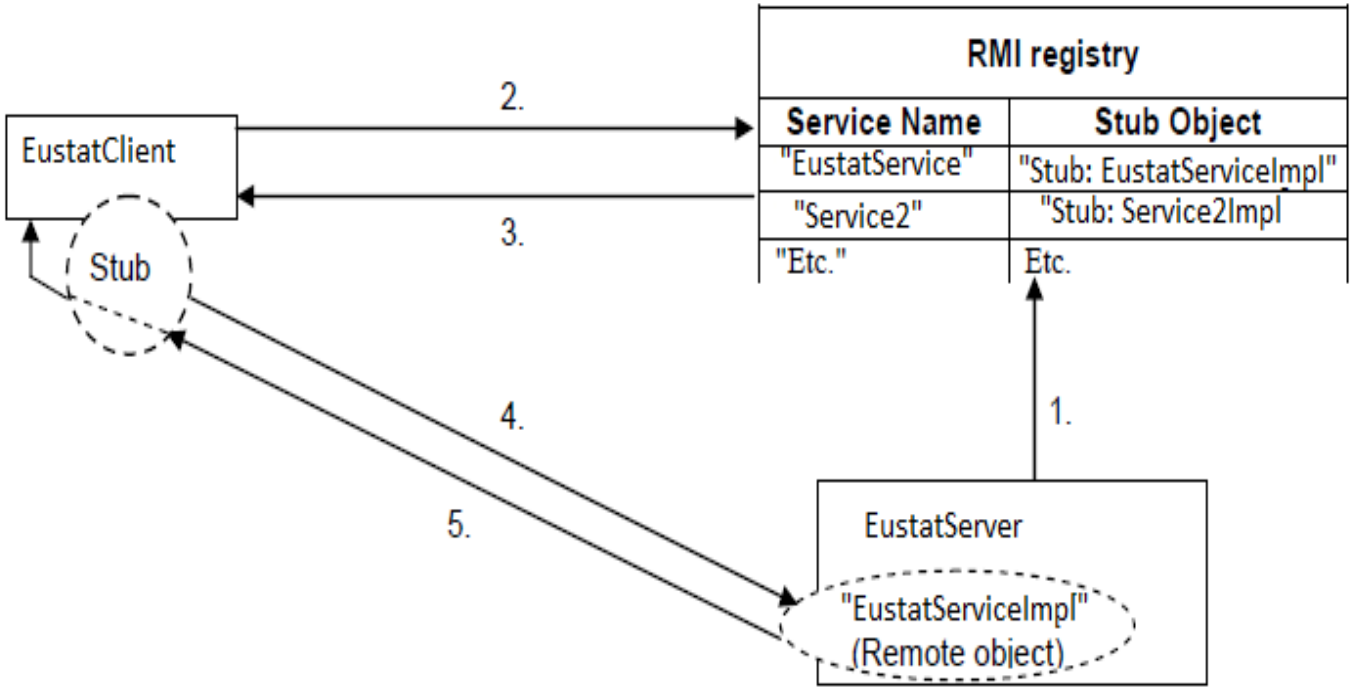
1. `EutStatServer` genera lo stub per l'oggetto remoto `EustatServiceImpl` e lo registra nel `Registry RMI` con il nome: "`EustatService`"
2. `EustatClient` effettua una look up nel `Registry` (`Naming.lookup(...)`)

JAVA RMI > 1.5: THE EUSTAT APPLICATION



1. `EutStatServer` genera lo stub per l'oggetto remoto `EustatServiceImpl` e lo registra nel `Registry RMI` con il nome: "`EustatService`"
2. `EustatClient` effettua una look up nel `Registry` (`Naming.lookup(...)`)

JAVA RMI > 1.5: THE EUSTAT APPLICATION



3. il registry RMI restituisce lo stub al client

4. il client `EustatClient` effettua l'invocazione di metodo remoto mediante lo stub

5. viene restituito al client il servizio richiesto

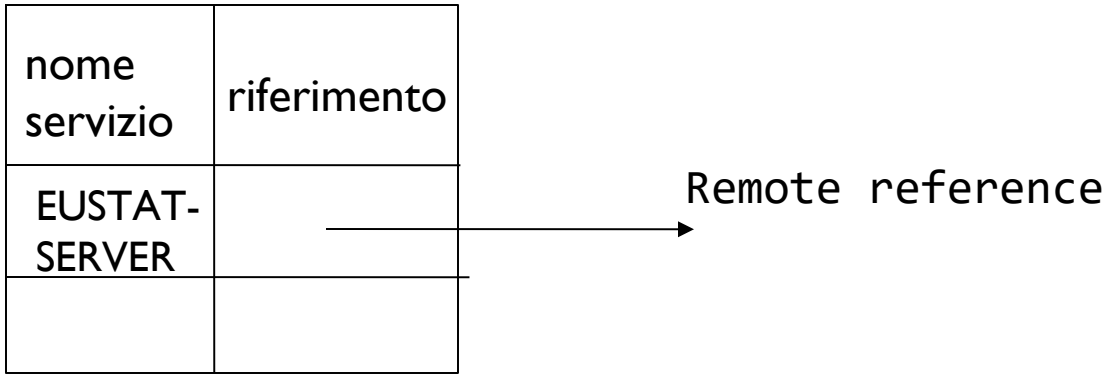
RMI: IL REGISTRY

JAVA mette a disposizione del programmatore un semplice dervizio di naminh (registry) che consente la registrazione ed il reperiento dello Stub

`Registry` :

- un servizio in ascolto sulla porta indicata.
- simile ad un DNS per oggetti remoti, contiene legami tra il nome simbolico dell'oggetto remoto ed il riferimento all'oggetto
- supporre Registry in esecuzione su local host:

```
LocateRegistry.createRegistry(args[0]);
Registry r=LocateRegistry.getRegistry(args[0]);
```



RMI: IL REGISTRY

- per creare e gestire direttamente da programma oggetti di tipo `Registry`, utilizzare la classe **`LocateRegistry`**, alcuni dei metodi statici implementati

```
public static Registry createRegistry(int port)
public static Registry getRegistry(String host, int port)
```

- `createRegistry`: lanciare un servizio di registry `RMI` sull'host locale, su una porta specificata e restituisce un riferimento al registro

- `getRegistry` reperisce e restituisce un riferimento ad un registro `RMI` su un certo host ad una certa porta.
  - ottenuto il riferimento al registro `RMI` , si possono invocare i metodi definiti dall'interfaccia Registry.
  - solo allora viene creata una connessione col registro.
  - restituita una eccezione se non esiste il registro `RMI` corrispondente

REGISTRY: UNA SOLUZIONE ALTERNATIVA

è anche possibile attivare un servizio di registry da shell, invece che da programma:

```
> rmiregistry & (in LINUX)
> start rmiregistry (in WINDOWS)
```

- viene attivato un registry associato per default alla porta 1099
- se la porta è già utilizzata, viene sollevata un'eccezione. Si può anche scegliere esplicitamente una porta

```
> rmiregistry  2048 &
```

- RMI fornisce la classe `java.rmi.Naming` per interagire con un registro preventivamente lanciato da linea di comando col comando

IL CLIENT RMI

- per accedere all'oggetto remoto, il client deve ricercare lo `Stub` dell'oggetto remoto
- accede al `Registry` attivato sul server effettuando una ricerca con il nome simbolico dell'oggetto remoto.
- il riferimento restituito è un riferimento allo stub dell'oggetto
  - se si usano le reflection, viene restituito anche il codice dello Stub
  - altrimenti ricerca nel classpath e quindi nel codebase
- il riferimento restituito è di tipo generico `Object:` è necessario effettuarne il casting al tipo definito nell'interfaccia remota
- invoca i metodi dell'oggetto remoto come fossero metodi locali (l'unica differenza è che occorre intercettare `RemoteException`)

IL CLIENT RMI

```
import java.rmi.*;
public class EUStatsClient {
public static void main (String args[]) {
    EUStats serverObject;
    Remote RemoteObject;
    /* Check number of arguments */
    /* If not enough, print usage string and exit */
    if (args.length < 2) {
    System.out.println("usage: java EUStatsClient
                          port countryname");return;}
       /* Set up a security manager as before */
       /* System.setSecurityManager(new  RMISecurityManager());
```

IL CLIENT RMI

```
try {Registry r = LocateRegistry.getRegistry(args[0]);
     RemoteObject = r.lookup("EUSTATS-SERVER");
     serverObject = (EUStats) RemoteObject;
     System.out.println("Main language(s) of " + args[1] + "
          is/are " + serverObject.getMainLanguages(args[1]));
     System.out.println("Population of " + args[1] + " is "
                + serverObject.getPopulation(args[1]));
     System.out.println("Capital of " + args[1] + " is "
                + serverObject.getCapitalName(args[1]));}
catch (Exception e) {
    System.out.println("Error in invoking object method " +
                          e.toString() + e.getMessage());
    e.printStackTrace();}}}
```

IL CLIENT RMI

- Gli argomenti passati al client da riga di comando sono il nome del paese europeo di cui si vogliono conoscere alcune informazioni e la porta su cui è in esecuzione il servizio di Registry.
- Remote indica un oggetto remoto su cui si deve fare il casting al tipo definito dalla interfaccia `EUStats`
- NOTA BENE: l'invocazione dei metodi remoti avviene mediante lo stesso meccanismo utilizzato per l'invocazione dei metodi locali
- Compilazione ed invocazione del client

```
Javac  EUStats.java  Javac EUStatsClient.java
Java   EUStatsClient ip-host countryname
```

- NOTA BENE: quando viene compilato il codice del client, il compilatore JAVA ha bisogno di accedere al file .class corrispondente all'interfaccia, `EUStats.class`

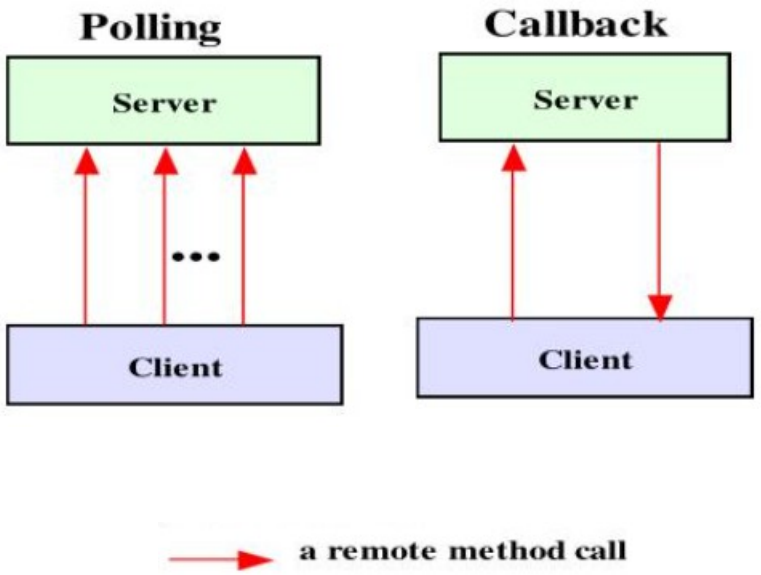
CALL BACK: INTRODUZIONE

- nel modello client server, il server è passivo: attende l'arrivo di richieste e fornisce risposte (servizi)
- alcune applicazioni richiedono che il server inizi la comunicazione, quando avvengono certi eventi
  - monitoring
  - multiplayer games: ogni gicatore viene notificato dal server quando lo stato del gioco cambia
  - aste: ogni volta che un utente fa una nuova offerta, il server avverte tutti i partecipanti all'asta
  - chat-room: avvertire gli utenti quando un nuovo utente entra nel gruppo.
  - message/bullettin board

POLLING VERSO CALLABACK

polling

- il client interroga periodicamente, mediante un metodo RMI, il server (passivo) per verificare l'occorrenza di un evento
- svantaggio
  - utilizzo non efficiente delle risorse del sistema
  - alto costo

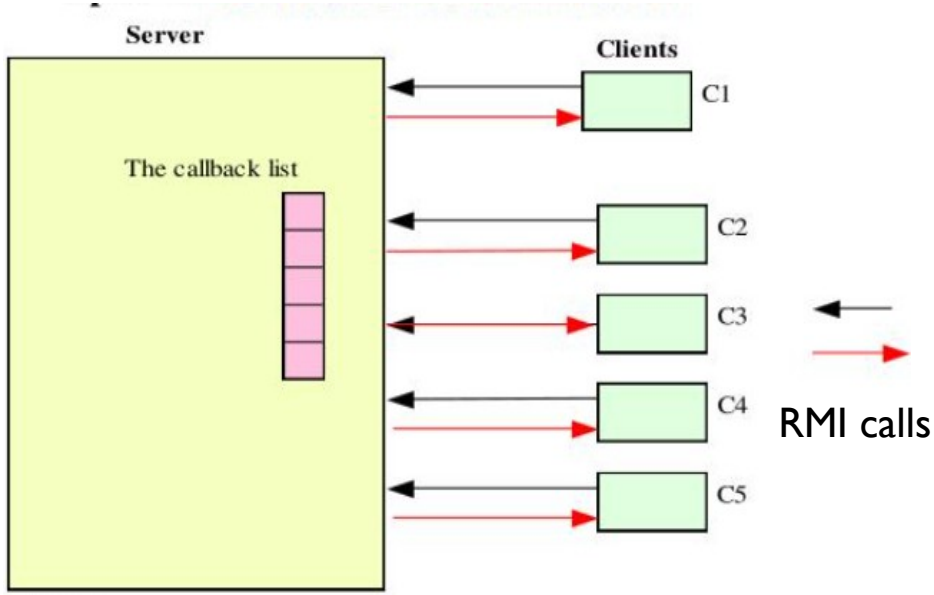


callback

- il client registra il suo interesse in un evento
- il server (attivo) notifica l'evento, quando questo si verifica,

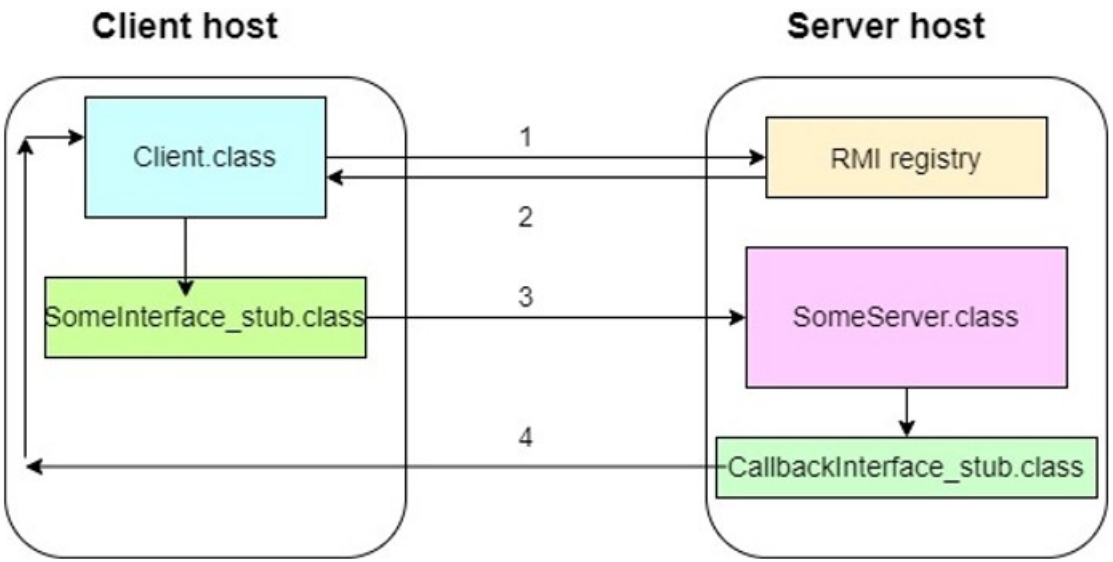
MECCANISMI DI NOTIFICA: CALLBACKS

- il client registra sul server il suo interesse per un evento
- il server notifica tale evento, quando questo occorre, ad ogni client registrato
  - vantaggi:
    - il client non si blocca: la notifica è asincrona
    - uso efficiente delle risorse



- noto anche come paradigma publish-subscribe
- realizza il pattern `Observer` in ambiente distribuito

CALLBACK IN BREVE



1. il client cerca il riferimento al server nel registry
2. il registry restituisce un riferimento all'oggetto remoto
3. attraverso lo stub, il client invoca il metodo remoto che gli consente di registrare la callback. Il server registra la callback (lo stub passato dal client) nella sua callback list
4. quando avviene l'evento il server invia una callback ad ogni client registrato usando gli stub registrati nella callabck list

IMPLEMENTAZIONE DELLE CALLBACK

- il server definisce un'interfaccia remota `ServerInterface` con un metodo remoto utilizzato dal client per registrare la callback
- il client definisce un'interfaccia remota `ClientInterface` con un metodo remoto utilizzato dal server per notificare l'evento al client
- il client reperisce il riferimento all'oggetto remoto tramite il `registry`
- il server riceve dal client
  - lo stub della callback, ovvero lo stub dell'oggetto remoto del client che contiene il metodo per la notifica
  - lo stub è passato come parametro del metodo di registrazione della callback
  - il server non utilizza il registry per individuare l'oggetto remoto del client

CALLBACK VIA RMI: IL SERVER

- definisce un oggetto remoto `ROS` che implementa `ServerInterface`
  - contiene il metodo per la registrazione della callback
  - parametro del metodo di registrazione: riferimento allo stub del client
- quando riceve una invocazione del metodo remoto
  - riceve `ROC`, riferimento all'oggetto remoto del client
  - lo memorizza in una propria struttura dati
- al momento della notifica, utilizza `ROC` per invocare il metodo remoto sul client

CALLBACK VIA RMI: IL CLIENT

- definisce un oggetto remoto ROC che implementa `ClientInterface`
  - contiene un metodo che consente la notifica dell'evento atteso.
  - questo metodo verrà invocato dal server.
- ricerca l'oggetto remoto ROS del server che contiene il metodo per la registrazione mediante un servizio di `Registry`
- al momento della registrazione sul server, passa al server lo stub di ROC
- non registra l'oggetto remoto ROC in un registro
- la notifica asincrona viene implementata mediante due invocazioni remote sincrone

CALLBACK: UN ESEMPIO

Un server gestisce le quotazioni di borsa di un titolo azionario. Ogni volta che si verifica una variazione del valore del titolo, vengono avvertiti tutti i client che si sono registrati per quell'evento.

Il server definisce un oggetto remoto che fornisce metodi per

- consentire al client di registrare/cancellare una callback
- avvertire i client registrati quando si verifica una variazione sul titolo

Il client vuole essere informato quando si verifica una variazione

- registra una callback presso il server
- aspetta per un certo intervallo di tempo durante cui riceve le variazioni di quotazione
- alla fine cancella la registrazione della propria callback presso il server

L'INTERFACCIA DEL CLIENT

```
import java.rmi.*;
public interface NotifyEventInterface extends Remote {
    /* Metodo invocato dal  server per notificare un evento ad un
       client remoto. */
    public void notifyEvent(int value) throws
                                                RemoteException;}
```

`notifyEvent(int value)` è il metodo

- esportato dal client
- utilizzato dal server per la notifica di un nuova quotazione del titolo azionario

L'INTERFACCIA DEL CLIENT: IMPLEMENTAZIONE

```
import java.rmi.*;
import java.rmi.server.*;
public class NotifyEventImpl extends RemoteObject implements
                                        NotifyEventInterface {
  /* crea un nuovo callback client */
  public NotifyEventImpl( ) throws RemoteException
        { super( ); }

  /* metodo che può essere richiamato dal servente per notificare una   nuova
     quotazione del titolo */
  public void notifyEvent(int value) throws RemoteException {
        String returnMessage = "Update event received: " + value;
        System.out.println(returnMessage); }
        }
```

LANCIO DEL CLIENT

```
import java.rmi.*; import java.rmi.registry.*; import java.rmi.server.*;
import java.util.*;
public class Client {
   public static void main(String args[ ]) {
      try
         {System.out.println("Cerco il Server");
          Registry registry = LocateRegistry.getRegistry(5000);
          String name = "Server";
          ServerInterface server =
                        (ServerInterface) registry.lookup(name);
          /* si registra per la callback */
          System.out.println("Registering for callback");
          NotifyEventInterface callbackObj =  new NotifyEventImpl();
          NotifyEventInterface stub = (NotifyEventInterface)
                             UnicastRemoteObject.exportObject(callbackObj, 0)
          server.registerForCallback(stub);
```

ATTIVAZIONE DEL CLIENT

```
// attende gli eventi generati dal server per
// un certo intervallo di tempo;
// che rappresenta l'intervallo di tempo in cui vuole ricevere le notifiche
// dal server
   Thread.sleep (20000);
/* cancella la registrazione per la callback */
   System.out.println("Unregistering for callback");
    server.unregisterForCallback(stub);
  } catch (Exception e)
        { System.err.println("Client exception:"+ e.getMessage(  ));}
 } } } }
```

L'INTERFACCIA DEL SERVER

```
import java.rmi.*;
public interface ServerInterface extends Remote
   {
    /* registrazione per la callback */
     public void registerForCallback
        (NotifyEventInterface ClientInterface) throws RemoteException;

    /* cancella registrazione per la callback */
     public void  unregisterForCallback  (NotifyEventInterface
                        ClientInterface) throws RemoteException; }
```

IL SERVER: IMPLEMENTAZIONE

```
import java.rmi.*; import java.rmi.server.*; import java.util.*;
public class ServerImpl extends RemoteObject implements ServerInterface
    { /* lista dei client registrati */
     private List <NotifyEventInterface> clients;
      /* crea un nuovo servente */
     public ServerImpl()throws RemoteException
          {super( );
           clients = new ArrayList<NotifyEventInterface>( ); };
     public synchronized void registerForCallback
                    (NotifyEventInterface ClientInterface) throws RemoteException
          {if (!clients.contains(ClientInterface))
              { clients.add(ClientInterface);
               System.out.println("New client registered." );}};
```

IL SERVER: IMPLEMENTAZIONE

```
/* annulla registrazione per il callback */
  public synchronized void unregisterForCallback (NotifyEventInterface Client)
                                                        throws RemoteException
            {if (clients.remove(Client))
                   {System.out.println("Client unregistered");}
            else { System.out.println("Unable to unregister client."); }
               }

 /* notifica di una variazione di valore dell'azione
 /* quando viene richiamato, fa il callback a tutti i client
    registrati */
  public void update(int value) throws RemoteException
                                {doCallbacks(value);};
```

IL SERVER: IMPLEMENTAZIONE

```
private synchronized void doCallbacks(int value) throws
                                                    RemoteException
        { System.out.println("Starting callbacks.");
          Iterator i = clients.iterator( );
          //int numeroClienti = clients.size( );
          while (i.hasNext()) {
               NotifyEventInterface client =
                                   (NotifyEventInterface) i.next();
               client.notifyEvent(value);
           }
          System.out.println("Callbacks complete.");}
       }
```

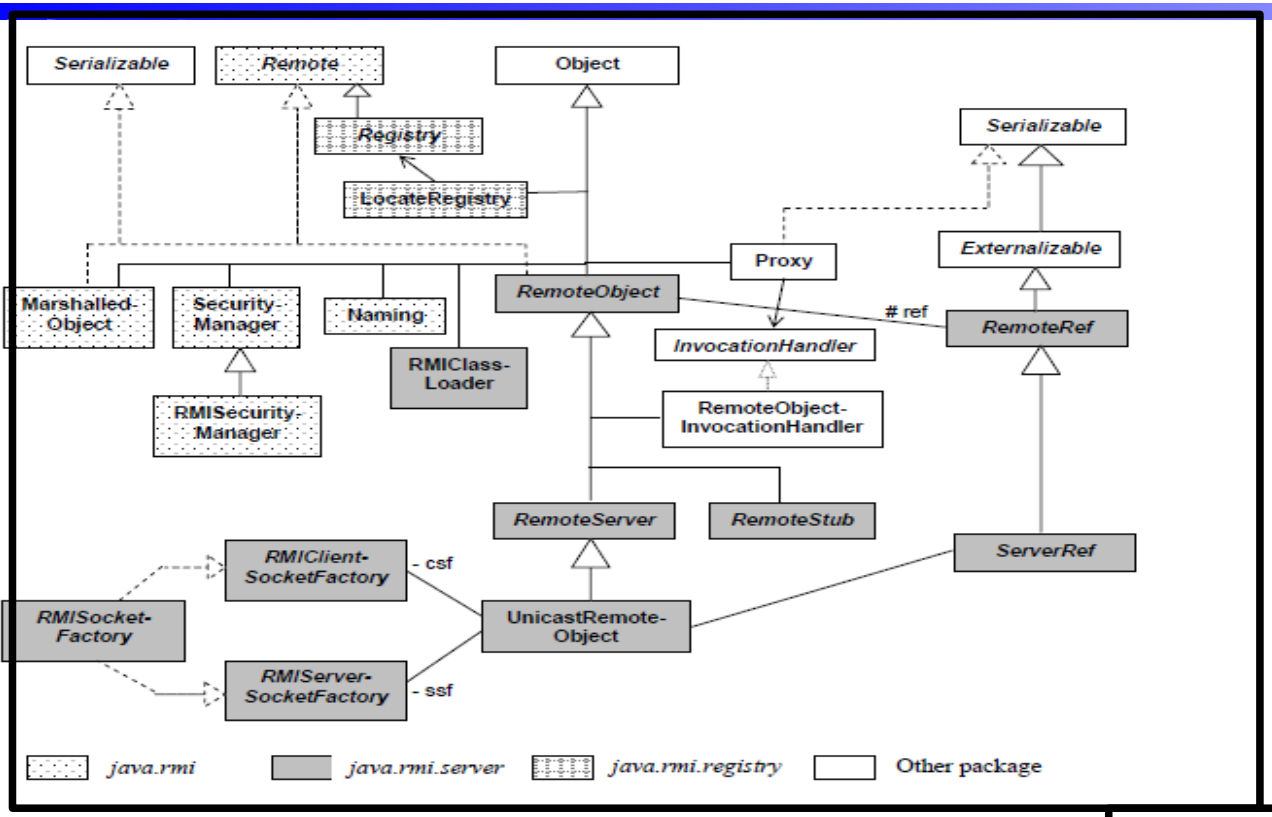
ATTIVAZIONE DEL SERVER

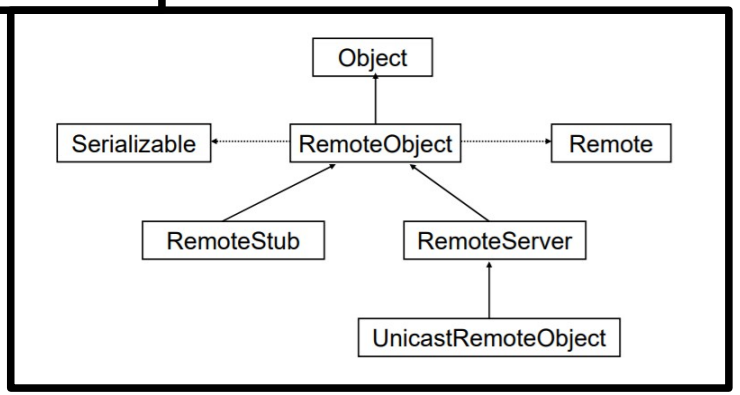
```
import java.rmi.server.*; import java.rmi.registry.*;
public class Server   {
        public static void main(String[  ] args) {
           try{ /*registrazione presso il registry */
                ServerImpl server =  new ServerImpl( );
                ServerInterface stub=(ServerInterface)
                        UnicastRemoteObject.exportObject (server,39000);
                String name = "Server";
                LocateRegistry.createRegistry(5000);
                Registry registry=LocateRegistry.getRegistry(5000);
                registry.bind  (name, stub);
             while (true) { int val=(int) (Math.random( )*1000);
                        System.out.println("nuovo update"+val);
                        server.update(val);
                        Thread.sleep(1500);}
               } catch (Exception e) { System.out.println("Eccezione" +e);}}}
```

RMI CALLBACKS: RIASSUNTO

- Il client crea un oggetto remoto, oggetto callback ROC, che implementa un'interfaccia remota che deve essere nota al server
- Il server definisce un oggetto remoto ROS, che implementa una interfaccia remota che deve essere nota al client
- Il client reperisce ROS mediante il meccanismo di lookup di un registry
- ROS contiene un metodo che consente al client di registrare il proprio ROC presso il server
- quando il server ne ha bisogno, reperisce un riferimento ad ROC dalla struttura dati in cui lo ha memorizzato al momento della registrazione e contatta il client via RMI

RMI: GERARCHIA DELLE CLASSI





ASSIGNMENT : GESTIONE CONGRESSO

Si progetti un'applicazione Client/Server per la gestione delle registrazioni ad un congresso. L'organizzazione del congresso fornisce agli speaker delle varie sessioni un'interfaccia tramite la quale iscriversi ad una sessione, e la possibilità di visionare i programmi delle varie giornate del congresso, con gli interventi delle varie sessioni.

Il server mantiene i programmi delle 3 giornate del congresso, ciascuno dei quali è memorizzato in una struttura dati come quella mostrata di seguito, in cui ad ogni riga corrisponde una sessione (in tutto 12 per ogni giornata). Per ciascuna sessione vengono memorizzati i nomi degli speaker che si sono registrati (al massimo 5).

Sessione	Intervento 1	Intervento 2	…	…	Intervento 5
S1	Nome Speaker1	Nome Speaker2			
S2					
S3					
…					
S12					

ASSIGNMENT: GESTIONE CONGRESSO

Il client può richiedere operazioni per:

- registrare uno speaker ad una sessione;
- ottenere il programma del congresso;

Il client inoltra le richieste al server tramite il meccanismo di RMI. Prevedere, per ogni possibile operazione una gestione di eventuali condizioni anomale (ad esempio la richiesta di registrazione ad una giornata e/o sessione inesistente oppure per la quale sono già stati coperti tutti gli spazi d'intervento)

Il client è implementato come un processo ciclico che continua a fare richieste sincrone fino ad esaurire tutte le esigenze utente. Stabilire una opportuna condizione di terminazione del processo di richiesta.