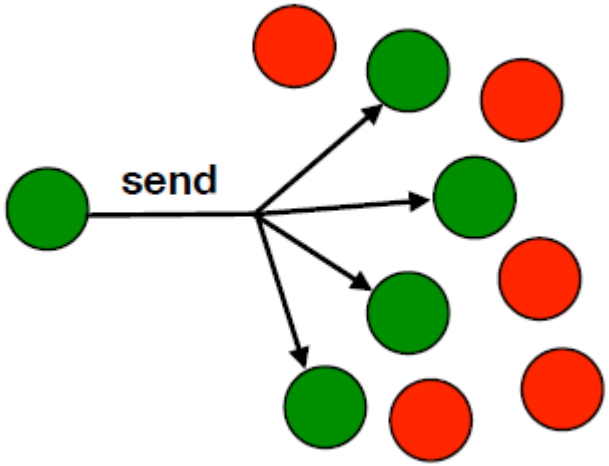
Laboratorio di Reti
Lezione 10
Multicast
Remote Method Invocation
JAVA RMI

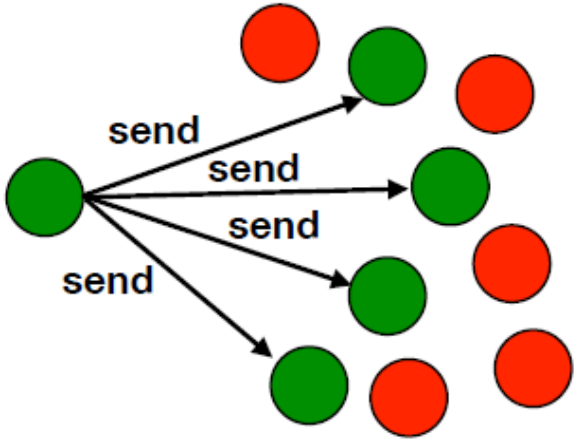
26/11/2020

Laura Ricci

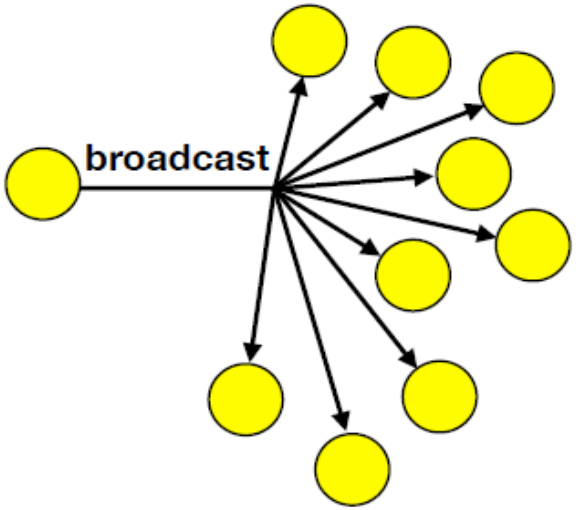
PARADIGMI DI COMUNICAZIONE



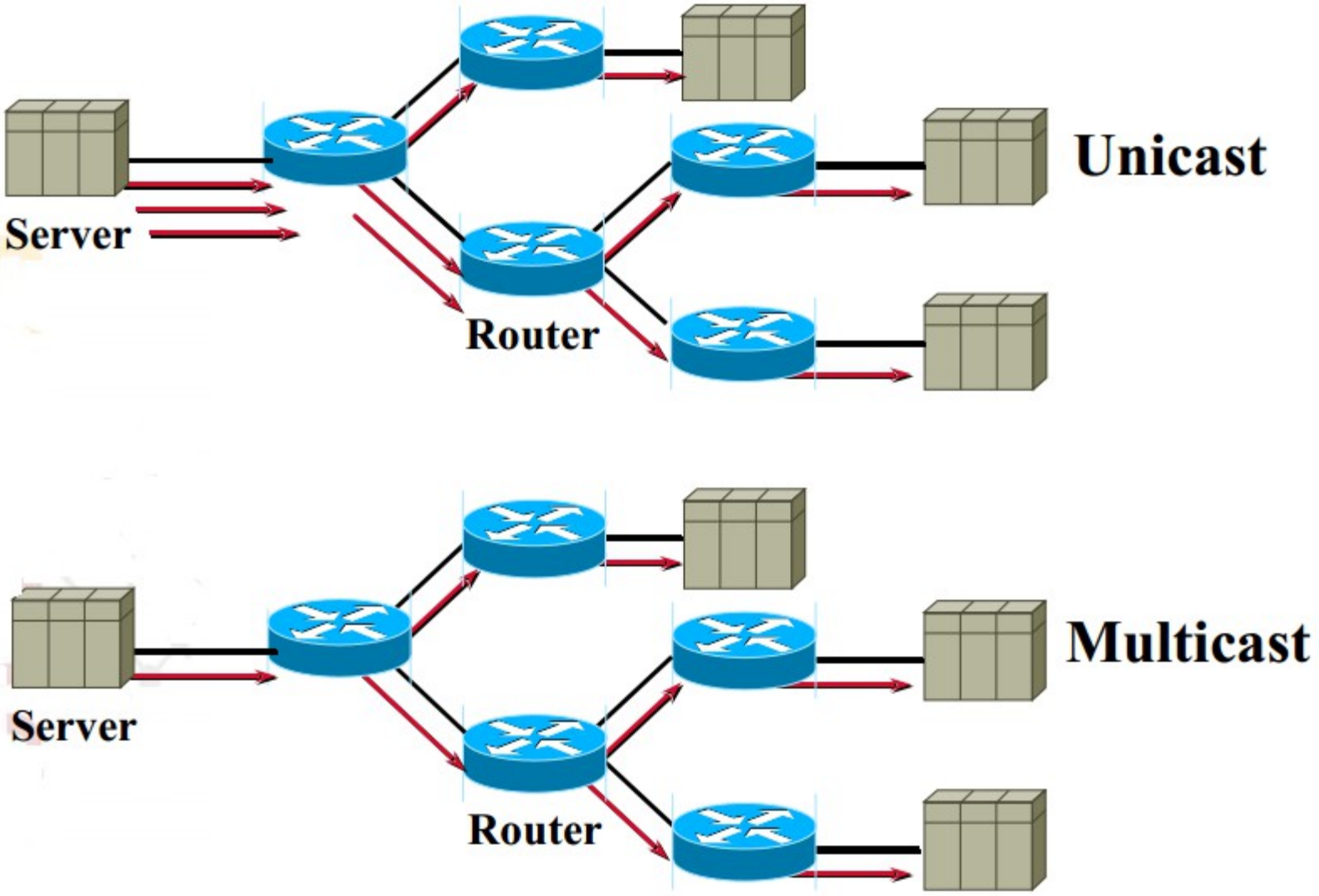
Multicast



Unicast

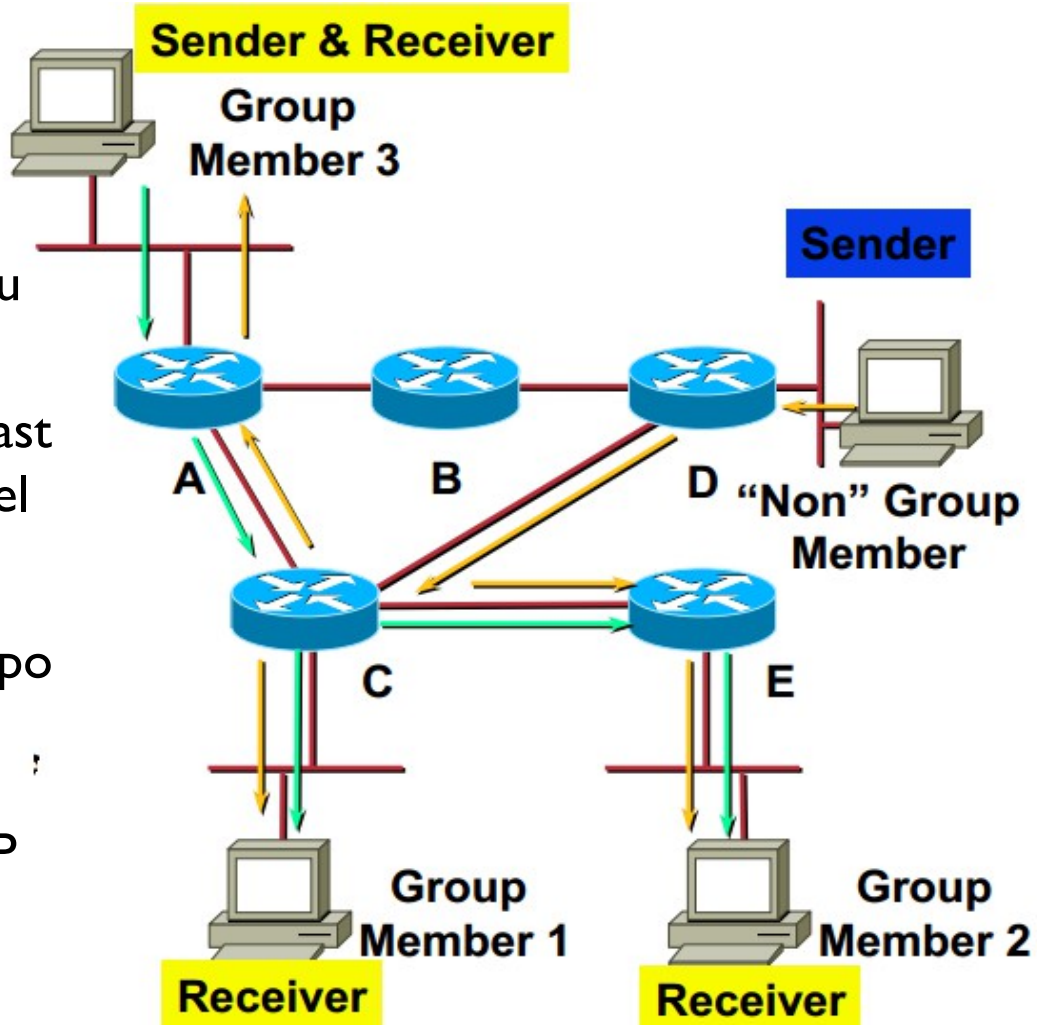


UNICAST VERSO MULTICAST



GRUPPI MULTICAST

- IP multicast basato sul concetto di gruppo
  - insieme di processi in esecuzione su host diversi
- tutti i membri di un gruppo di multicast ricevono un messaggio spedito su quel gruppo
- non occorre essere membri del gruppo per inviare i messaggi su di esso
- gestiti a livello IP dal protocollo IGMP



MULTICAST API

deve contenere almeno primitive per:

- unirsi ad un gruppo di multicast
- lasciare un gruppo
- spedire messaggi ad un gruppo
  - il messaggio viene recapitato a tutti i processi che fanno parte del gruppo in quel momento
- ricevere messaggi indirizzati ad un gruppo

Il supporto deve fornire

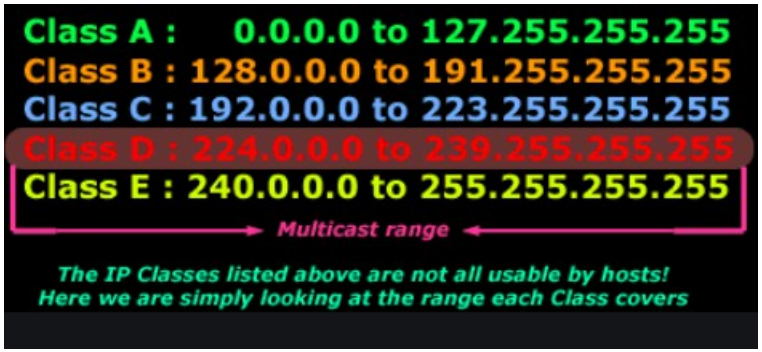
- uno schema di indirizzamento per identificare univocamente un gruppo.
- un meccanismo che registri la corrispondenza tra un gruppo ed i suoi partecipanti (IGMP)
- un meccanismo che ottimizzi l'uso della rete nel caso di invio di pacchetti ad un gruppo di multicast

SCHEMA DI INDIRIZZAMENTO MULTICAST

- basato sull'idea di riservare un insieme di indirizzi IP per il multicast

- *IPV4*: indirizzo di un gruppo è un indirizzo in classe D
  - [224.0.0.0 – 239.255.255.255]

- *IPV6*: tutti gli indirizzi di multicast iniziano con FF
  - riservati da IANA



- i primi 4 bit del primo ottetto = 1110
- i restanti bit identificano il particolare gruppo

MULTICAST ADDRESSING E SCOPING

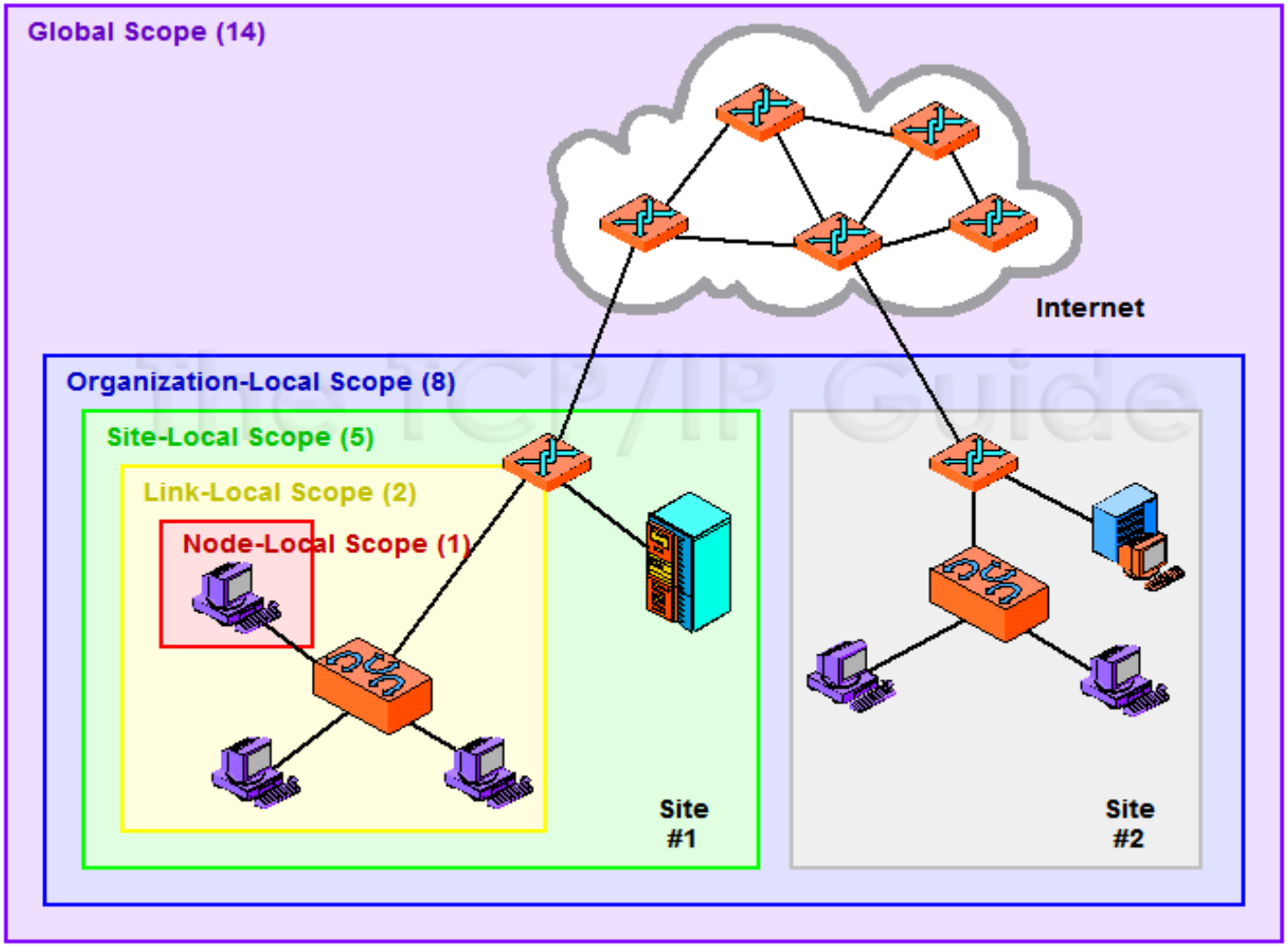
- Multicast addressing: come scegliere un indirizzo di multicast?
  - indirizzo multicast: deve essere noto "collettivamente" a tutti i partecipanti al gruppo
  - indirizzi statici: scelti da una autorità per un certo servizio
  - indirizzi dinamici: come evitare collisioni?

- Multicast scoping (scope: portata, raggio): come limitare la diffusione di un pacchetto?
  - TTL scoping: il TTL limita la diffusione del pacchetto
  - administrative scoping: a seconda dell'indirizzo in classe D scelto, la diffusione del pacchetto viene limitata ad una parte della rete i cui confini sono definiti da un amministratore di rete

TTL SCOPING

- IP Multicast Scoping: limita la diffusione di un pacchetto multicast
- utilizza il TTL;
  - ad ogni pacchetto IP viene associato un valore rappresentato su un byte, riferito come `TTL (Time-To-Live)` del pacchetto
  - valori del `TTL` nell'intervallo `0-255`
  - indica il numero massimo di routers attraversati dal pacchetto
  - il pacchetto viene scartato dopo aver attraversato `TTL` routers
  - associazione valori di `TTL`-aree geografiche
    - `1, 16, 63,e 127`, rappresentano la sottorete locale, la rete della organizzazione di cui fa parte l'host, la rete regionale e la rete globale

ADMINISTRATIVE SCOPING



ADMINISTRATIVE SCOPING

Address type	IPv4	IPv6
Multicast	224.0.0.0 to 239.255.255.25	Begins with byte FF
MC global	224.0.1.0 to 238.255.255.255	FF0E or FF1E
Org MC	239.192.0.0/14	FF08 or FF18
MC site-local	N/A	FF05 or FF15
MC link-local	224.0.0.0	FF02 or FF12
MC node local	127.0.0.0	FF01 or FF11
Private	10.0.0.0 to 10.255.255.255 172.16.0.0 to 172.31.255.255 192.168.0.0 to 192.168.255.255	fd00::/8

ADMINISTRATIVE SCOPING

```
import java.net.*;
public class IndirizziIP {
public static void main (String args[]) throws Exception
    { InetAddress address=InetAddress.getByName(args[0]);
      if (address.isMulticastAddress()){
         if (address.isMCGlobal()){
            System.out.println("e' un indirizzo di multicast globale");}
        else if (address.isMCOrgLocal())
              {System.out.println ("e' un indirizzo organization local");}
        else if (address.isMCSiteLocal())
           {System.out.println ("e' un indirizzo site local");}
        else if(address.isMCLinkLocal())
              {System.out.println("e' un indirizzo link local");}; }
             else System.out.println("non e' un indirizzo di
                                                    Multicast");}}
```

INDIRIZZAMENTO GRUPPI DI MULTICAST

- l'allocazione degli indirizzi di multicast su Internet è una procedura molto complessa, che prevede un ampio numero di casi

- Gli indirizzi possono essere
  - statici: assegnati da una autorità di controllo, utilizzati da particolari protocolli/ applicazioni.
    - l'indirizzo rimane assegnato a quel gruppo, anche se, in un certo istante non ci sono partecipanti
  - dinamici: si utizzano protocolli particolari che consentono di evitare che lo stesso indirizzo di multicast sia assegnato a due gruppi diversi
    - esistono solo fino al momento in cui esiste almeno un partecipante
    - richiedono un protocollo per l'assegnazione dinamica degli indirizzi

INDIRIZZI DI MULTICAST STATICI

- gli indirizzi statici possono essere assegnati
  - da IANA

    https://www.iana.org/assignments/multicast-addresses/multicast-addresses-xml
  - dall'amministratore di rete

- assegnati da IANA
  - sono validi per tutti gli host della rete e possono essere 'cablati'

    nel codice delle applicazioni
  - ad esempio l'indirizzo di multicast 224.0.1.1 è assegnato al network

    time protocol, protocollo utilizzato per sincronizzare i clocks di più hosts
- assegnati dall'amministratore di una certa organizzazione
  - valgono per tutti gli host della rete amministrata

INDIRIZZI DI MULTICAST DINAMICI

- per ottenere un indirizzo di multicast in modo dinamico, è necessario utilizzare un protocollo opportuno

- nell'ambito di una sottorete gestita mediante un unico dominio amministrativo
  - *Multicast Address Dynamic Client Allocation Protocol (MADCAP).*

- nell'ambito più generale della rete
  - *Multicast Address Set Claim (MASC), SSM , SDR*

MULTICAST: CARATTERISTICHE

- utilizza il paradigma connectionless (UDP):
  - richieste n×(n-1) connessioni per un gruppo di n applicazioni, se si usa la comunicazione connection oriented
- comunicazione connectionless adatta per il tipo di applicazioni verso cui è orientato il multicast
  - trasmissione di dati video/audio: invio dei frames di una animazione.
  - è più accettabile la perdita occasionale di un frame piuttosto che un ritardo costante tra la spedizione di due frames successivi
- esistono librerie JAVA non standard che forniscono multicast affidabile
  - garantiscono che il messaggio venga recapitato una sola volta a tutti i processi del gruppo
  - possono garantire altre proprietà relative all'ordinamento con cui i messaggi spediti al gruppo di multicast vengono recapitati ai singoli partecipanti

JAVA API PER MULTICAST

`MulticastSocket`:

- estende `DatagramSocket`
- socket su cui ricevere i messaggi da un gruppo di multicast
- effettua overriding dei metodi esistenti in `DatagramSocket` e fornisce nuovi metodi per l'implementazione di funzionalità tipiche del multicast

```
import java.net.*;
import java.io.*;
public class multicast
    {public static void main (String [  ] args)
          {try
               {MulticastSocket ms = new MulticastSocket( );}
          catch (IOException ex) {System.out.println("errore"); }
          }}
```

JAVA API PER MULTICAST

```
import java.net.*;
import java.io.*;
public class multicast
{public static void main (String [ ] args)
    {  try {MulticastSocket ms = new MulticastSocket(4000);
            InetAddress ia=InetAddress.getByName("226.226.226.226");
            ms.joinGroup (ia);
            }
      catch (IOException ex) {System.out.println("errore"); }}}
```

- `joinGroup` necessaria nel caso si vogliano ricevere messaggi dal gruppo di multicast
- lega il multicast socket ad un gruppo di multicast: tutti i messaggi ricevuti tramite quel socket provengono da quel gruppo
- `IOException` sollevata se l'indirizzo di multicast è errato

JAVA API PER MULTICAST

```
import java.io.*;  import java.net.*;
public class provemulticast {
public static void main (String args[]) throws Exception
    { byte[] buf = new byte[10];
      InetAddress     ia = InetAddress.getByName("228.5.6.7");
      DatagramPacket  dp = new DatagramPacket (buf,buf.length);
      MulticastSocket ms = new MulticastSocket (4000);
      ms.joinGroup(ia);
      ms.receive(dp);    } }
```

- se attivo due istanze di provemulticast sullo stesso host (la reuse socket settata true) non viene sollevata una `BindException`
- l'eccezione verrebbe invece sollevata se si utilizzasse un `DatagramSocket`
- servizi diversi in ascolto sulla stessa porta di multicast
- non esiste una corrispondenza biunivoca porta-servizio

JAVA API PER MULTICAST

- ogni socket multicast ha una proprietà, la reuse socket, che se settata a true, da la possibilità di associare più socket alla stessa porta

- nelle ultime versioni è possibile impostarne il valore

```
try   {sock.setReuseAddress(true);}
catch (SocketException se) {…}
```

- nelle prime versioni di JAVA la proprietà era settata per default a true

MULTICAST SNIFFER

Dopo aver ricevuto in input il nome simbolico di un gruppo di multicast si unisce al gruppo e 'sniffa' i messaggi spediti su quel gruppo, stampandone il contenuto

```
import java.net.*; import java.io.*;
public class multicastsniffer {
public static void main (String[] args)
    {InetAddress group = null;
     int port = 0;
     try{
        group = InetAddress.getByName(args[0]);
        port = Integer.parseInt(args[1]);
        } catch(Exception e){System.out.println("Uso:java
              multicastsniffer multicast_address port");
      System.exit(1); }
```

MULTICAST SNIFFER

```
MulticastSocket ms=null;
try{ ms = new MulticastSocket(port);
     ms.joinGroup(group);
     byte [ ] buffer = new byte[8192];
     while (true)
     { try
        {DatagramPacket dp=new DatagramPacket(buffer,buffer.length);
         ms.receive(dp);
         String s = new String(dp.getData());
         System.out.println(s);
        }
      catch (IOException ex){System.out.println (ex);}
      finally{ if (ms!= null) {
             try { ms.leaveGroup(group);
                   ms.close();
                 } catch (IOException ex){} }}}}
```

SPEDIRE PACCHETTI

Per spedire messaggi ad un gruppo di multicast:

- creare un `DatagramSocket` su una porta anonima
- non è necessario collegare il socket ad un gruppo di multicast
- creare un pacchetto inserendo nell'intestazione l'indirizzo del gruppo di multicast a cui si vuole inviare il pacchetto
- spedire il pacchetto tramite il socket creato

```
public void send (DatagramPacket p) throws IOException
```

SPEDIRE PACCHETTI

```
import java.io.*;
import java.net.*;
public class multicast {
public static void main (String args[])
{try
    { InetAddress ia=InetAddress.getByName("228.5.6.7");
      byte [  ] data;
      data="hello".getBytes();
      int port= 6789;
      DatagramPacket dp = new DatagramPacket(data,data.length,ia,port);
      DatagramSocket ms = new DatagramSocket(6789);
      ms.send(dp);
      Thread.sleep(80000);
    } catch(IOException ex){ System.out.println(ex);
    }}}
```

TIME TO LIVE

TTL Scoping, implementazione

- il mittente specifica un valore per del TTL per i pacchetti spediti
- il TTL viene memorizzato in un campo dell'header del pacchetto IP
- TTL viene decrementato da ogni router attraversato
- se TTL = 0, il pacchetto viene scartato

Valore impostato = 1 (i pacchetti di multicast non possono lasciare la rete locale)

Per modificare il valore di default: posso associare il TTL al multicast socket

```
MulticastSocket s = new MulticastSocket();
s.setTimeToLive(1));
```

OLTRE I SOCKETS....

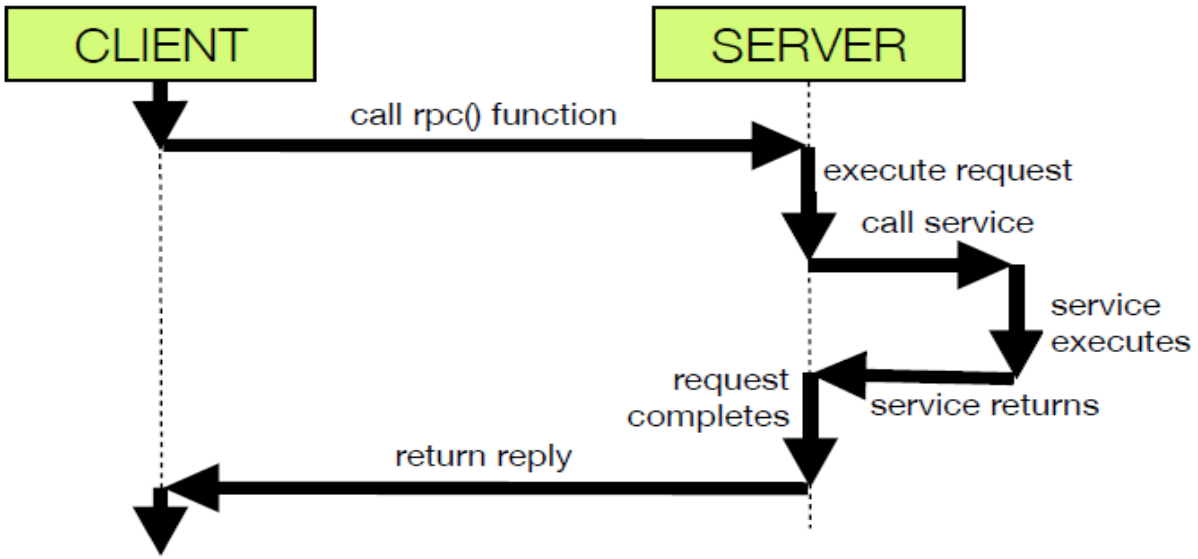
- un'applicazione JAVA distribuita:
  - è composta da computazioni eseguite su JVM differenti,
  - possibilmente in esecuzione su host differenti comunicanti tra loro (un'applicazione multithreaded non è distribuita !)

- il meccanismo dei socket è flessibile e potente per la programmazione di applicazioni distribuite, ma è di basso livello:
  - richiede la progettazione di veri e propri protocolli di comunicazione e la verifica non banale delle loro funzionalità
  - la serializzazione consente di ridurre complessità dei protocolli inviando dati strutturati

- un'alternativa è utilizzare una tecnologia di più alto livello, originariamente indicata come Remote Procedure Call (RPC), quindi evoluta in RMI (Remote Procedure Call)
  - interfaccia di comunicazione rappresentata dall'invocazione di una procedura/metodo remoto, invece che da un socket

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



Ad esempio: richiesta di un servizio di stampa e restituzione esito operazione

REMOTE PROCEDURE CALL: UN ESEMPIO

paradigma di interazione tra client e server a domanda/risposta:

- un client
  - richiede ad un server la stampa di un messaggio.
    - il server restituisce un codice che indica l'esito della operazione
  - attende l'esito dell'operazione
  - la richiesta del client al server è implementata come una 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

- il programmatore non si deve più preoccupare di sviluppare protocolli per il trasferimento, la verifica, la codifica/decodifica dei dati.
  - operazioni interamente gestite dal supporto
  - utilizzo di stub o proxy (alla lettera moncone, mozzicone) "rappresentanti del server" sul client

- limiti della tecnologia RPC (SUN RPC):
  - parametri e risultati devono avere tipi primitivi
  - la programmazione è essenzialmente procedurale
  - la localizzazione del server non è trasparente (il client deve conoscere l'IP e la porta su cui il server è in esecuzione)
  - non basata sulla programmazione ad oggetti, quindi mancano i concetti di ereditarietà, incapsulamento, polimorfismo, …

- trasferimento del paradigma RPC verso il paradigma ad oggetti distribuiti

REMOTE METHOD INVOCATION

A partire dagli inizi degli anni '90 sono state proposte diverse tecnologie, per superare i limiti di RPC:

- CORBA: sviluppato con l'obiettivo di supportare applicazioni scritte in linguaggi diversi su piattaforme differenti.
  - obiettivo: inter-operabilità
  - l'"object model" di riferimento deve essere "language neutral"
- DCOM (Distributed component Object Module): supporta applicazioni scritte in linguaggi differenti, ma su piattaforme Win32. Esistono delle implementazioni per sistemi Unix.
- .NET remoting**:** supporta applicazioni scritte in linguaggi differenti, su piattaforma Windows.
- Web Service/SOAP**:** un framework per oggetti remoti basato su http e XML
- Java RMI: supporta applicazioni JAVA distribuite, ovvero le applicazioni possono essere distribuite su differenti JVM: ambiente omogeneo

JAVA RMI: GENERALITA'

- consente di avere oggetti remoti attivabili, ovvero servizi che si attivano "on demand", cioè a seguito di una invocazione, e che si disattivano quando non utilizzati.

- trasparenza:
  - l'utilizzo di oggetti remoti risulta largamente trasparente: una volta localizzato l'oggetto, il programmatore utilizza i metodi dell'oggetto come se questi fosse locale
  - codifica, decodifica, verifica, e trasmissione dei dati sono effettuati dal supporto RMI in maniera completamente trasparente all'utente.

- supporta le specifiche IIOP (Internet InterORB Protocol) che consentono l'integrazione tra JAVA RMI ed altri framework basati sul paradigma ad oggetti

JAVA RMI: SEMANTICA DEGLI OGGETTI

- attualmente l'implementazione di JAVA prevede oggetti:
  - di tipo unicast: ad un certo istante esiste una sola istanza dell'oggetto remoto, nella rete
  - volatili: il tempo di vita di un oggetto è al più pari a quello del processo che l'ha creato
  - non-rilocabili: l'oggetto remoto non può essere rilocato in un altro processo
- invocazione del metodo deve tenere in conto diverse possibilità di errore
  - client o server crash, perdite di dati sulla rete,...
- diversi tipi di semantica:
  - at least once: se il client riceve una risposta del server, il metodo può essere stato eseguito più di una volta: ad esempio, se ha subito un crash prima di inviare la risposta
  - at most once (JAVA RMI)
  - exactly once....

JAVA RMI: GENERALITA'

- Un oggetto remoto è un oggetto i cui metodi possono essere acceduti da un diverso spazio di indirizzamento
  - una JVM diversa
  - che è 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, JDBC, …
- 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: ARCHITETTURA AD ALTO LIVELLO

- l'architettura RMI prevede le seguenti entità
  - registry
  - client
  - server

- principali operazioni da effettuare diverse rispetto all'uso di un oggetto locale:
  - il server deve esportare gli oggetti remoti, il client deve individuare un riferimento all'oggetto remoto.
  - registry permette il bootstrap, mettendo in comunicazione client e server
    - è un servizio di naming che agisce da 'yellow pages'
  - server registra riferimenti agli oggetti remoti nel registry, tramite la bind
  - client cercano i riferimenti gli oggetti remoti nel registry
    - tramite la lookup, a partire dal nome pubblico dell'oggetto

RMI: ARCHITETTURA AD ALTO LIVELLO

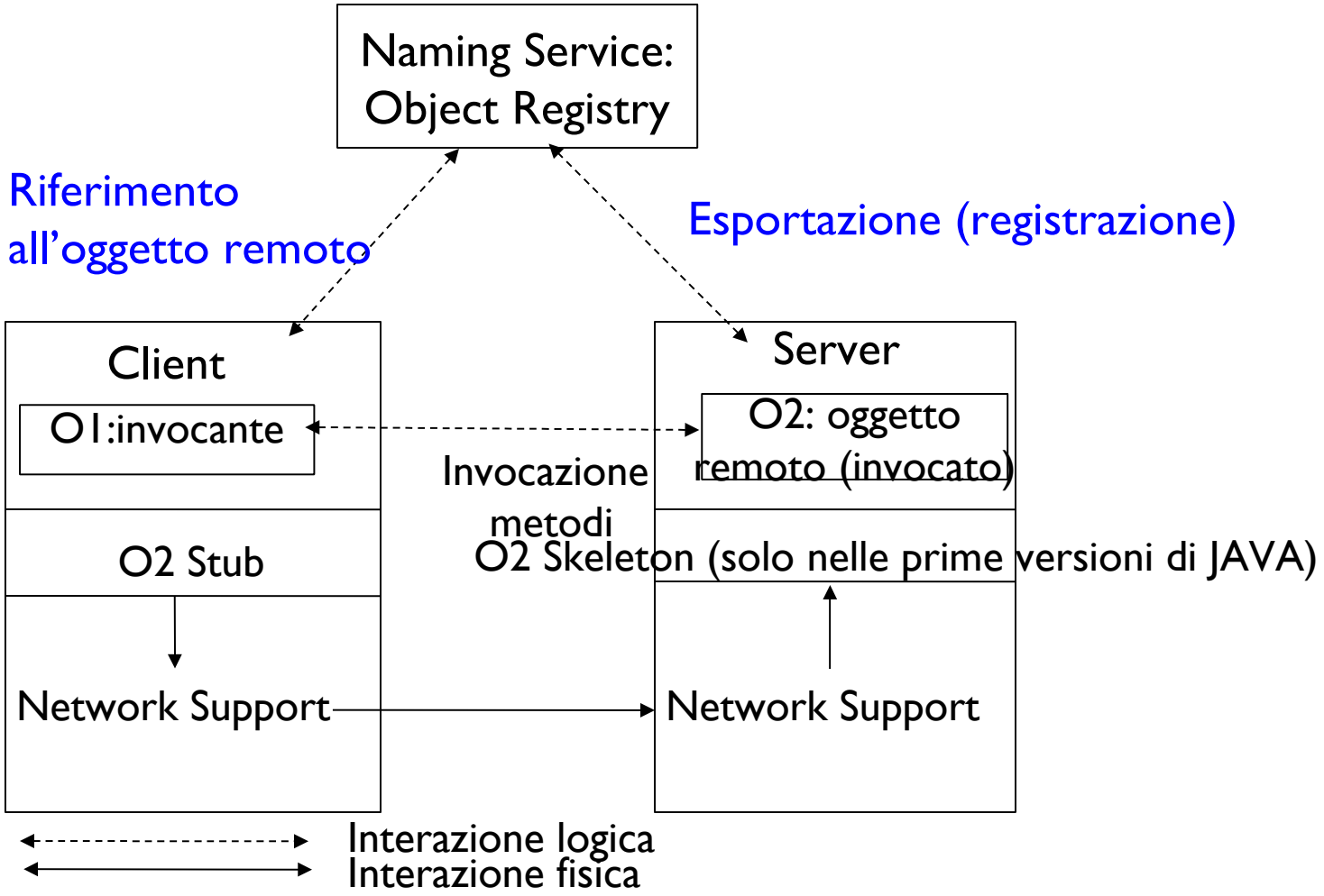
quando il client vuole accedere all'oggetto remoto:

- ricerca un riferimento all'oggetto remoto nel registry
- invoca il servizio mediante chiamate di metodi che sono le stesse delle invocazioni di metodi locali

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

- invocazione dei metodi di un oggetto remoto:
  - a livello logico: identica all' invocazione di un metodo locale
  - a livello di supporto: il supporto provvede
    - a trasformare i parametri della chiamata remota in dati da spedire sulla rete.
    - all'invio vero e proprio dei dati sulla rete

RMI: SCHEMA ARCHITETTURALE



RMI STEP BY STEP

Un esempio di applicazione che sfrutta RMI.

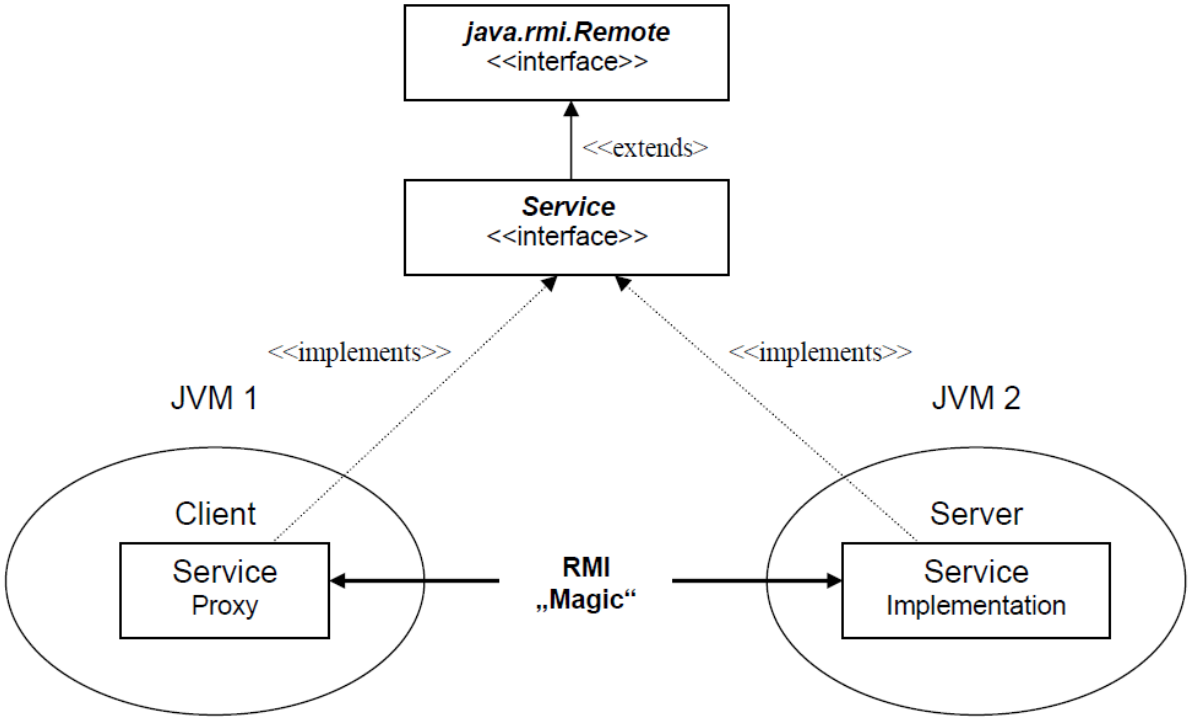
definire un server che implementi, mediante un oggetto remoto, il seguente servizio:

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



RMI : L'INTERFACCIA

- un'interfaccia è remota se e solo se:
  - estende `java.rmi.Remote` o un'altra interfaccia che estenda `java.rmi.Remote`
- `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 eccezioni remote, della classe `java.rmi.RemoteException` oppure di una sua superclasse (`java.io.IOException`, `java.lang.Exception`,...)

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

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 REMOTO

- implementazione l'interfaccia lato server
  - definire il codice dei metodi
- alcune differenze rispetto alla implementazione degli oggetti standard
- le operazioni base della classe `Object`
  - `equals`
  - `hascode`
  - `toString`

  devono essere ridefinite per gli oggetti remoti, perchè hanno una semantica diversa
- il package `Java.rmi.server` supporta la definizione degli oggetti remoti e da una ridefinizione standard delle operazioni precedenti

QUALE SEMANTICA PER GLI OGGETTI REMOTI?

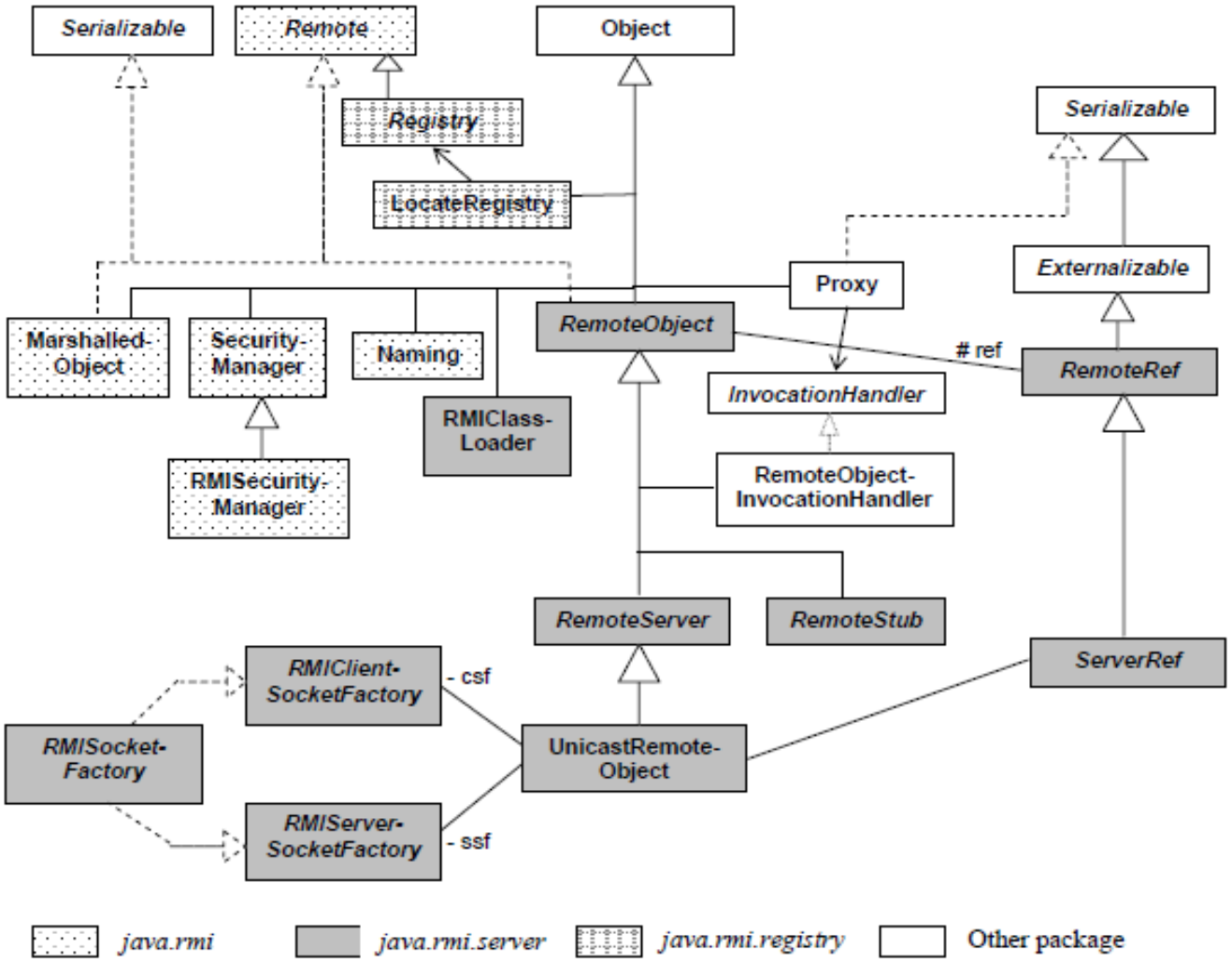
- Semantica del Metodo equals per oggetti remoti:
  - idea generale: due oggetti remoti sono considerati uguali se i loro riferimenti lo sono
  - "reference equality" per oggetti remoti: confronto tra riferimenti remoti.
  - "content equality" scartata dai progettisti perchè:
    - costo: implicherebbe chiamate remote
    - failure semantics: semantica diversa tra chiamate remote e chiamate locali del metodo equal
      - per oggetti remoti sarebbe stato necessario intercettare eccezioni `Remote`
      - non presenti nella segnatura del metodo `equal`

QUALE SEMANTICA PER GLI OGGETTI REMOTI?

- hashcode per oggetti remoti:
  - comportamento standard: calcola il valore hash dell'oggetto (utile ad esempio per `HashMap`)
  - restituisce lo stesso valore per tutte le referenze remote che puntano allo stesso `RemoteObject`.

- toString per oggetti remoti:
  - restituisce una rappresentazione sotto forma di stringa per l'oggetto remoto
  - ad esempio la stringa può includere host name + porta + identificatore dell'oggetto

RMI: PACKAGES E CLASSI



STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

- la semantica standard dei metodi ridefiniti per oggetti remoti è implementata nel package `java.rmi.server`

- classi del package `java.rmi.server`

  `UnicastRemoteObject < RemoteServer < RemoteObject < Object`

  dove A < B significa che A è una sottoclasse di B.

- la definizione dell'oggetto remoto può essere effettuata in modi diversi:
  - utilizzare la classe `RemoteObject` oppure `UnicastRemoteObject`:
    - effettua l'overriding di alcuni metodi della classe `Object` per adattarli al comportamento degli oggetti remoti
  - definire un oggetto come istanza della classe `Object`, per effettuare esplicitamente una ridefinizione di questi metodi (a carico del programmatore)

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

- l'implementazione dell'interfaccia può quindi fare uso delle classi descritte nelle slide precedenti, nei seguenti modi:

- Soluzione I

  definire una classe che implementi i metodi della interfaccia remota ed estenda la classe RemoteObject

```
public class MyServerRemoto extends RemoteServer implements IntRemota
    {public MyServerRemoto() throws RemoteException {  ..... }
          ......}
```

- vantaggi: si eredita la ridefinizione della semantica degli oggetti remoti definita da RemoteServer
- svantaggi
  - a causa dell'eredità singola, non si possono estendere altre classi
  - l'oggetto deve essere poi esportato esplicitamente

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

Soluzione 2

- definire una classe che implementi i metodi della interfaccia remota ed estenda la classe `UnicastRemoteObject`
  - estende `RemoteServer`, che estende, a sua volta, `RemoteObject`
  - contiene metodi per costruire ed esportare un oggetto remoto con semantica di tipo unicast, volatile, non migrabile.

```
public class MyServerRemoto extends UnicastRemoteObject
                                    implement IntRemota {
   public MyServerRemoto() throws RemoteException {
          super(); // E' qui che il server viene esportato
              .....}
```

- il costruttore di `UnicastRemoteObject`
  - esporta automaticamente l'oggetto remoto
  - crea automaticamente un server socket per ricevere le invocazioni di metodi remoti

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

Soluzione 3

- definire una classe che implementi i metodi della interfaccia remota senza estendere alcuna delle classi viste.

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

richiede esportazione esplicita

- vantaggi: possibilità di estendere un'altra classe utile per l'applicazione(ad esempio `MyClass`)
- svantaggi: semantica degli oggetti remoti demandata al programmatore (overriding metodi equals, hashcode,...)

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

- applichiamo i concetti visti alla implementazione del servizio `EUStatServer`
- utilizziamo una hash table per memorizzare i dati riguardanti le nazioni europee. In questo modo definiamo un semplice data base delle nazioni
  - utilizzata la classe `EUData` per definire oggetti (record) che descrivono la singola nazione
- definiamo una classe `EuStatServer` che, utilizzando la classe `EUData`, implementa l'oggetto remoto e ne crea un'istanza
  - implementa la interfaccia `EUStats`
  - il main della classe 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 il collegamento nel registry

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

Una classe di appoggio:

```
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 extends RemoteServer
                                        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( ); }
```

PASSO 3: GENERAZIONE STUB

- `Stub`
  - è un oggetto che consente di interfacciarsi con un altro oggetto (il target) in modo da sostituirsi ad esso
  - target: oggetto remoto
  - si comporta da intermediario: inoltra le chiamate che riceve al suo target
- 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

PASSO 3: ATTIVAZIONE SERVIZIO

- Il servizio viene creato allocando una istanza dell'oggetto remoto
- Il servizio viene attivato mediante:
  - creazione dell'oggetto remoto
  - registrazione dell'oggetto remoto in un registry

PASSO 3:ATTIVAZIONE DEL SERVIZIO

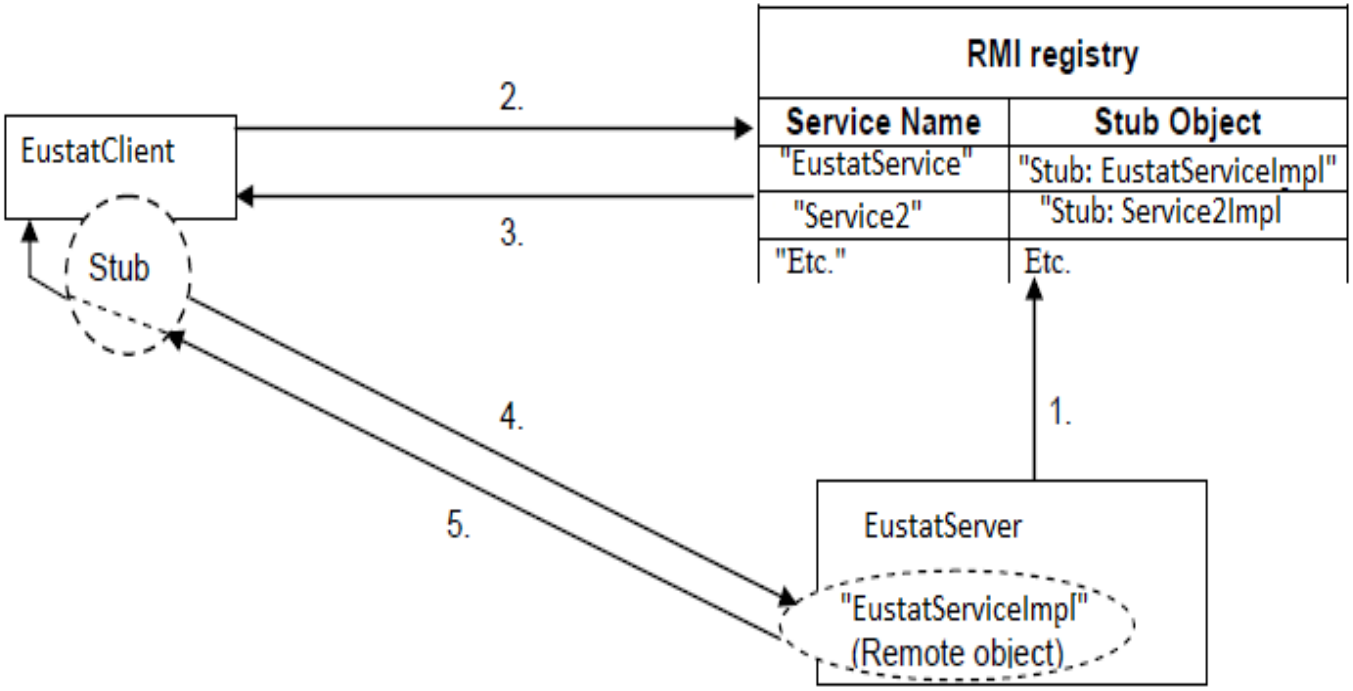
```
public static void main (String args[]) {

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`:

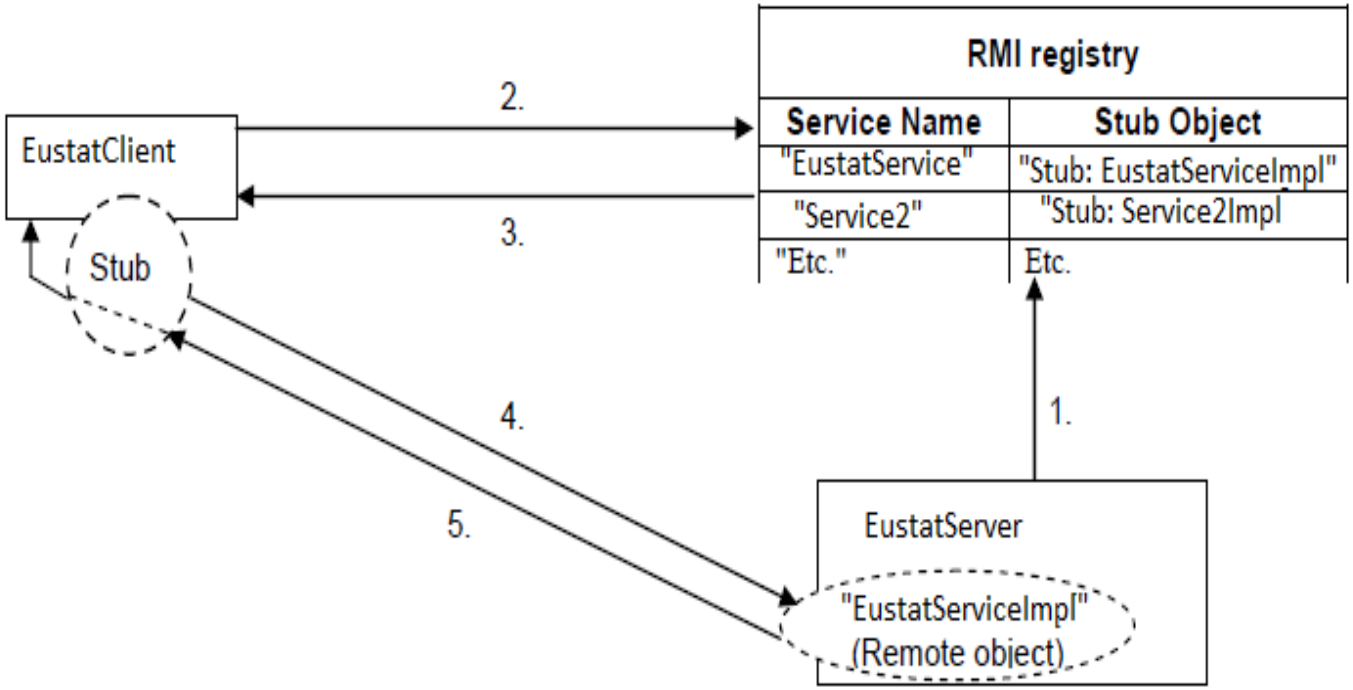
- crea un'istanza del servizio (oggetto remoto)
- invoca il metodo statico `UnicastRemoteObject.exportObject(obj,0)` che esporta dinamicamente l'oggetto,
- se si indica la porta 0 viene utilizzata una porta anonima scelta dal supporto
- restituisce un'istanza dell'oggetto, che "rappresenta" l'oggetto 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

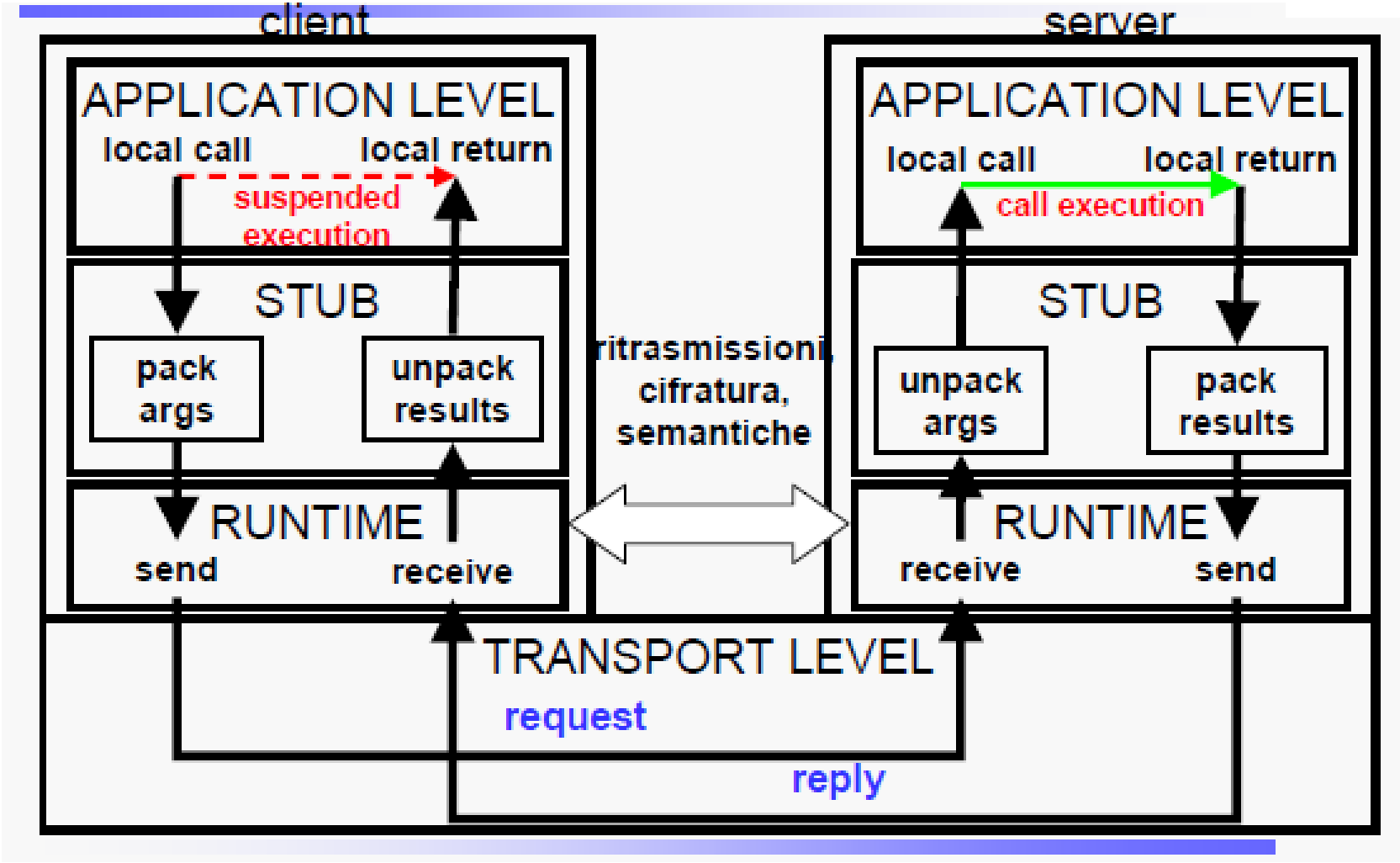


3. il registry RMI restituisce lo stub al client

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

5. viene restituito al client il servizio richiesto

RMI: SCHEMA ARCHITETTURALE



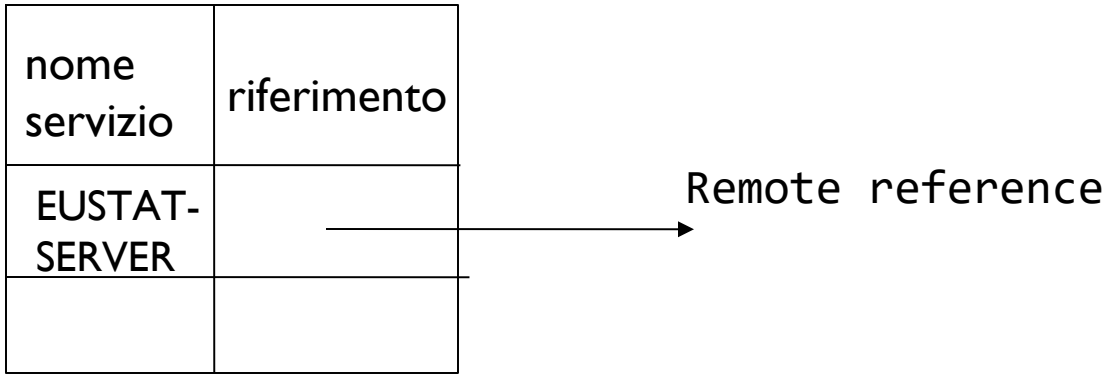
RMI: IL REGISTRY

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`

ESERCIZIO DI PREPARAZIONE ALL'ASSIGNMENT

Definire un Server WelcomeServer, che

- invia su un gruppo di multicast (welcomegroup), ad intervalli regolari, un messaggio di "welcome".
- attende tra un invio ed il successivo un intervallo di tempo simulato mediante il metodo `sleep( )`.

Definire un client `WelcomeClient` che si unisce a welcomegroup, riceve un messaggio di welcome, quindi termina.

* Ad esempio con indirizzo IP 239.255.1.3

ASSIGNMENT 1: MULTICAST DATE SERVER

Definire un Server `TimeServer`, che

- invia su un gruppo di multicast dategroup, ad intervalli regolari,la data e l'ora.
- attende tra un invio ed il successivo un intervallo di tempo simulata mediante il metodo `sleep( )`.

L'indirizzo IP di dategroup viene introdotto da linea di comando.

Definire quindi un client `TimeClient` che si unisce a dategroup e riceve, per dieci volte consecutive, data ed ora, le visualizza, quindi termina.

ASSIGNMENT 2: 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.

Nei primi assignment, eseguire client, server e registry sullo stesso host.