REMOTE PROCEDURE CALL (RPC)

estensione del normale meccanismo di chiamata a procedura, adatta per il modello cliente servitore

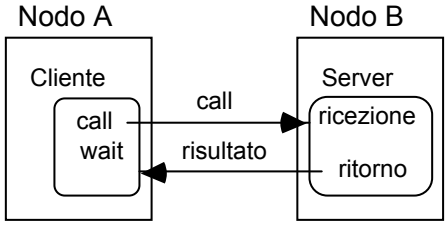
IDEA per uniformare
programmi concentrati e distribuiti

APPROCCIO linguistico:
il cliente invia la richiesta ed attende fino alla risposta del servitore stesso

A differenza della *chiamata a procedura locale*
- i processi **non** condividono lo spazio di indirizzamento
- i processi hanno vita **separata**
- possono accadere **malfunzionamenti**, sia ai nodi, sia alla interconnessione

RPC
consente di considerare anche il **controllo di tipo**
da livello di linguaggio del cliente al servitore

trattamento automatico degli argomenti di ingresso e di uscita dal cliente al servitore, e viceversa
marshalling



RPC

Birrel Nelson (1984)
usate in Xerox, Spice, Sun, HP, etc.

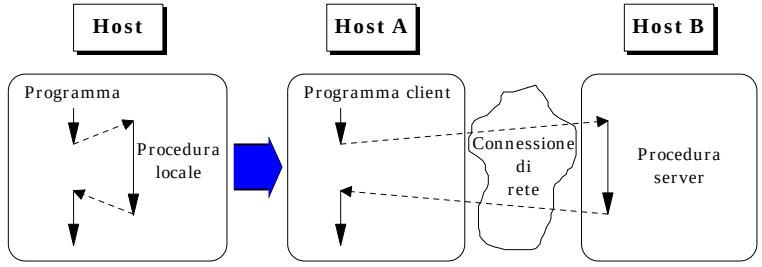
PROPRIETÀ
trasparenza approccio locale e remoto
uniformità totale è impossibile (guasti)
controllo di tipo e parametrizzazione
lo stesso controllo di tipo e dei parametri
controllo concorrenza e eccezioni
binding distribuito
possibile trattamento degli *orfani*
recovery in caso di fallimento: *orfano* chi resta

MODELLO CLIENTE/SERVITORE

invocazione di un servizio remoto

con **parametri tipati**, valore di **ritorno**

Modello di base: P.B. Hansen (Distributed Processing)
Nelson Birrel



RPC come astrazione

dello scambio messaggi o comunicazione

CLIENTE	SERVITORE
send (a chi)	**get-request** <operazione>
wait	**send-reply**

Progetto Athena MIT, ITC CMU, ...

PRIMITIVE

dalla parte del cliente

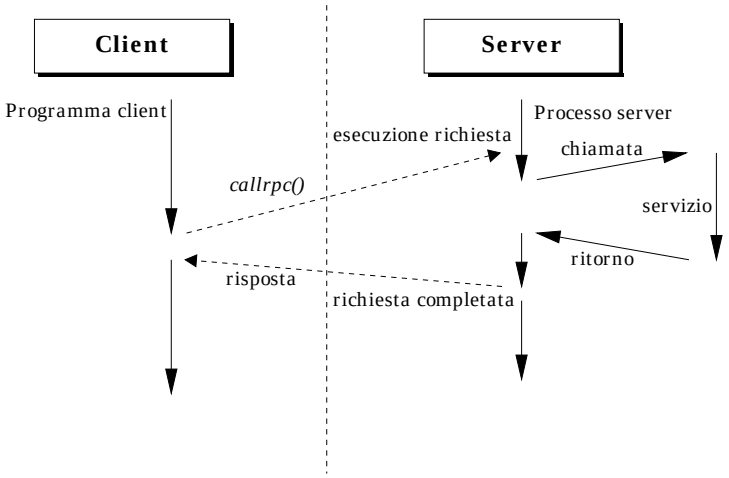
call servizio (nodoservitore, argomenti, risultato)

dalla parte del servitore

due possibilità:

- il servizio svolto da un unico **processo sequenziale**
- il servizio è un **processo indipendente**, generato per ogni richiesta (approccio implicito)

***Schema* NON TRASPARENTE**

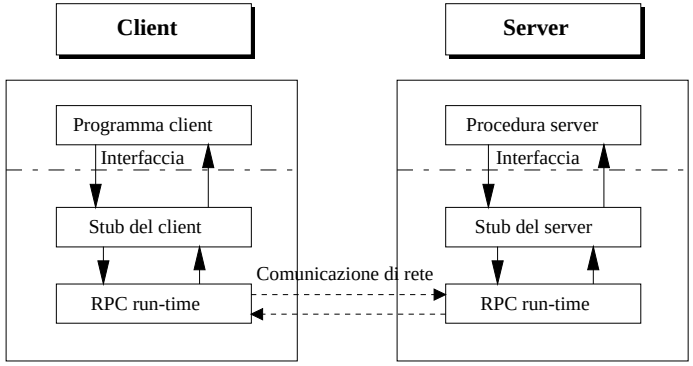


Primo modello con trasparenza

Birrel Nelson: uso di stub

ossia di **interfacce per la trasparenza**

che trasformano la richiesta da locale a remota



Il **cliente** invoca uno **stub**, che si incarica del trattamento dei parametri e della richiesta al supporto run-time, per il **trasporto** della richiesta

Il **servitore** riceve la richiesta dallo **stub** relativo, che si incarica del trattamento dei parametri dopo avere ricevuto la richiesta pervenuta dal **trasporto.** Al completamento del servizio, lo stub rimanda il risultato al cliente

Uso di STUB

Sono componenti che servono a superare i problemi di **trasparenza**

Lo sviluppo con STUB prevede **trasparenza**

gli stub sono in genere prodotti 'automaticamente'

L'UTENTE PROGETTA SOLO LE PARTI APPLICATIVE

GLI STUB

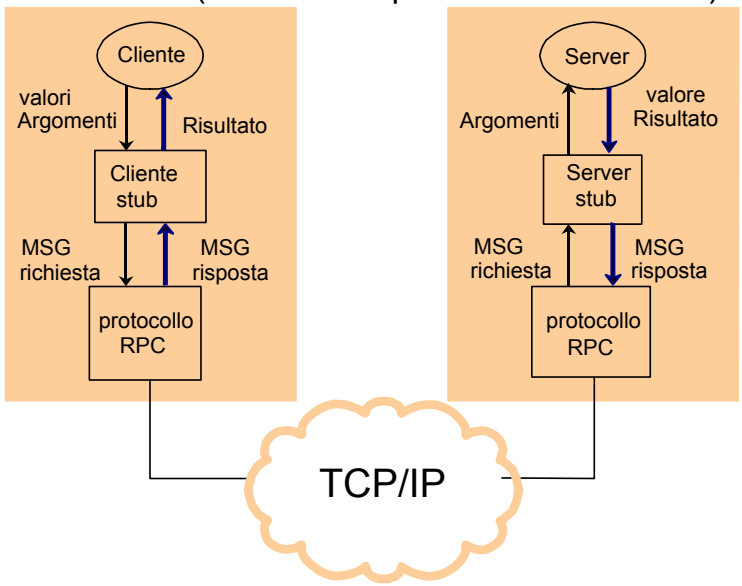
operazione(parametri)

stub cliente:
<ricerca del **servitore**>
<***marshalling*** argomenti>
<send richiesta>
<receive risposta>
<***unmarshalling*** risultato>
restituisci risultato
fine stub cliente;

stub servitore:
<attesa della richiesta>
<***unmarshalling*** argomenti>
invoca operazione locale
ottieni risultato
<***marshalling*** del risultato>
<send risultato>
fine stub servitore;

operazione(parametri)

In aggiunta, il controllo della **operazione** ed eventuali azioni di **ritrasmissione** (entro un tempo di durata massima)



Passaggio dei parametri

passaggio per **valore** o per **riferimento**

Trattamento dei parametri per valore

impaccamento dei parametri e disimpaccamento

marshalling ==> dipende dal linguaggio utilizzato
unmarshalling

Problema della rappresentazione dei dati

Se si vuole passare un **tipo primitivo** o un'entità con certi **valori privati**
marshalling e unmarshalling per la presentazione

Per un tipo utente costruito e dinamico, ad esempio, una **lista** o un **albero**
si deve muoverla e ricostituirla sul server
per poi riportarla sul nodo iniziale (?)
e la memoria dinamica (?)

Passaggio parametri

nel caso di **valore** ==> trasferimento e visita
(poi valore perso)
nel caso di **riferimento** ==>
uso di oggetti che rimangono nel nodo di partenza e devono essere identificati in modo unico nell'intero sistema
Se si vuole riferire un'**entità del cliente**
si passa il riferimento alla stessa entità
da riferire con RPC da nodi remoti

Per esempio, un **oggetto** del nodo di partenza che sia già in uso *deve potere essere riferito dal nodo servitore*

Passaggio dei parametri

passaggio per valore e per riferimento

nel caso di riferimento ==>
uso di oggetti confinati che sono identificati in modo unico nell'intero sistema

Trattamento dei parametri
impaccamento dei parametri e disimpaccamento

marshalling ==> dipende dal linguaggio utilizzato
unmarshalling

Exception handling

problemi tipici dipendenti dalla distribuzione e loro gestione

In un caso generale, una RPC può:
- produrre il servizio *con successo*
- produrre *insuccesso* e determinare una **eccezione**

In genere, si specifica la azione per il trattamento anomalo in un opportuno **gestore della eccezione**
Si possono prevedere più gestori a seconda dell'evento anomalo. Inoltre, si può anche inserire l'eccezione nello scope di linguaggio

CLU (Liskov) a livello di invocazione della RPC
MESA (Cedar) a livello di messaggio

AFFIDABILITÀ (Reliability)

Malfunzionamenti
- perdita di messaggio di richiesta o di risposta
- crash del nodo del cliente
- crash del nodo del servitore

In caso di crash del servitore, prima di fornire la risposta il **cliente** può:
- aspettare per sempre;
- **time-out e riportare una eccezione al cliente;**
- **time-out e ritrasmettere (uso identificatori unici);**

Operazioni **idempotenti**
che si possono eseguire un numero qualunque di volte con lo stesso esito

Semantica di funzionamento

may-be	time-out per il cliente
at-least-once	time-out e ritrasmissioni
at-most-once	tabelle delle azioni effettuate
exactly-once	e l'azione fatta fino alla fine

In caso di crash del **cliente**, si devono trattare **orfani**
- ***sterminio***: ogni orfano risultato di un crash viene distrutto
- ***terminazione a tempo***: ogni calcolo ha una scadenza, oltre la quale è automaticamente abortito
- ***reincarnazione (ad epoche)***: tempo diviso in epoche; tutto ciò che è relativo alla epoca precedente è obsoleto

SEMANTICA della comunicazione

MAY-BE UN SOLO INVIO

il messaggio può arrivare o meno

PROGETTO BEST-EFFORT IP UDP

non si fanno azioni per garantire affidabilità

AT-LEAST-ONCE PIÙ INVII AD INTERVALLI

il messaggio può arrivare anche **più** volte a causa della duplicazione dei messaggi dovuti a ritrasmissioni

==> semantica adatta per azioni **idempotenti**

*in caso di insuccesso **nessuna** informazione*

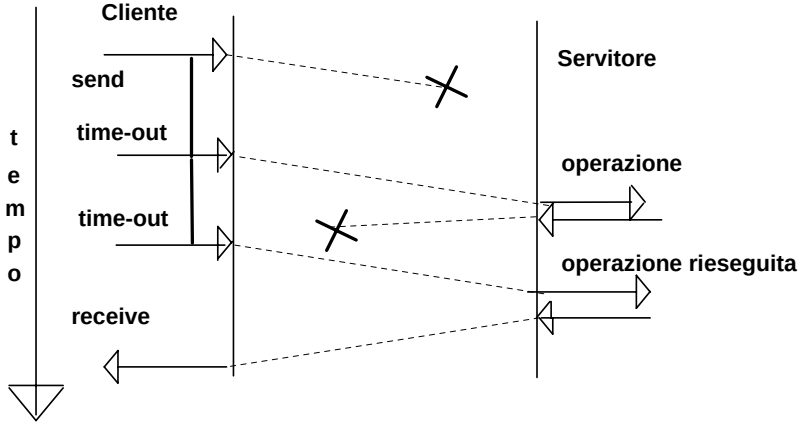
implementazione

PROGETTO RELIABLE (AL MITTENTE)

il cliente fa ritrasmissioni *(quante?, ogni quanto? ...)*

il server non se ne accorge

Semantica at-least-once



il cliente si preoccupa della affidabilità

Si noti la durata della azione

Il cliente decide (in modo unilaterale) la durata massima

AT-MOST-ONCE PIÙ INVII AD INTERVALLI STATO SUL SERVER

cliente e servitore lavorano coordinati per ottenere garanzie di correttezza e affidabilità

il messaggio, se arriva, viene considerato **al più** una volta

==> semantica che non mette vincoli sulle azioni conseguenti

*in caso di insuccesso **nessuna** informazione*

implementazione **TCP**

PROGETTO RELIABLE per cliente e servitore

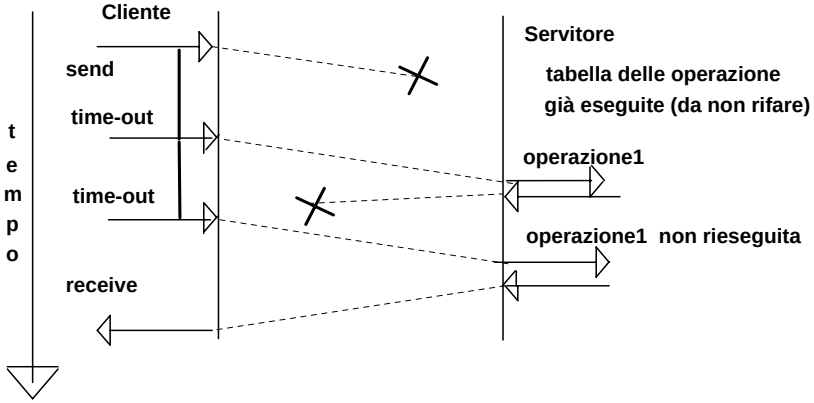
il cliente fa ritrasmissioni *(quante?, ogni quanto? ...)*

il server mantiene uno stato

per riconoscere i messaggi già ricevuti e

per non eseguire azioni più di una volta

Semantica at most-once



STATO MANTENUTO PER UN CERTO TEMPO

Si noti la durata della azione delle due parti

Il cliente decide la durata massima della propria azione

Il server mantiene uno stato per garantire correttezza

Per quanto tempo i pari mantengono lo stato della interazione? E se uno fallisce?

Semantica per at-most e at-least

Se le **cose vanno male manca il coordinamento e non sappiamo cosa sia successo**

EXACTLY-ONCE O ATOMICITÀ

Al termine sappiamo se l'operazione è stata fatta o meno

i pari lavorano entrambi per ottenere il massimo dell'accordo e della reliability

il messaggio arriva **una volta sola** oppure

il messaggio **o è arrivato** o **non è** stato **considerato** da entrambi

==> semantica molto coordinata sullo stato

PROGETTO con conoscenza dello stato finale
AFFIDABILITÀ e COORDINAMENTO massimo
semantica TUTTO o NIENTE

In caso le **cose vadano bene**
il messaggio arriva una volta e una volta sola viene trattato, riconoscendo i duplicati (tutto)

In caso le **cose vadano male**
il cliente e il servitore sanno se il messaggio è arrivato (e considerato 1 volta sola - *tutto*) o se non è arrivato
Se il messaggio non è arrivato, il tutto può essere riportato indietro (*niente*)

Completo coordinamento delle azioni
Durata delle azioni non predicibile

Se uno dei due fallisce, bisogna aspettare che abbia fatto il recovery (o qualcuno aspetta al suo posto)
Entrambi sanno realmente come è andata (tutto o niente)

FASI

compilazione
binding
trasporto
controllo
rappresentazione dei dati

problemi in ambiente eterogeneo

Le scelte sono diverse
scelta pessimistica e statica

La compilazione potrebbe risolvere ogni problema e forzare un **binding statico**

scelta ottimistica e dinamica

Il **binding dinamico** consente di ridirigere le richieste sul gestore più scarico o presente in caso di sistema dinamico

L'uso della **comunicazione** è intrinseco
tanto più veloce, tanto meglio

Il **controllo** consente anche di usare gli stessi strumenti per funzioni diverse, con maggiore asincronicità e maggiore complessità

necessità di **traslazione** dei dati
tanto più veloce, tanto meglio ***bilanciata*** con
la ridondanza che viene ritenuta necessaria

Trattamento del binding

legame tra cliente e servitore

STATICO vs. DINAMICO

due fasi:

- **servizio,** il cliente specifica a chi vuole essere connesso, come nome del servizio (NAMING)
- **indirizzo,** il cliente deve essere collegato al servitore che fornisce il servizio (ADDRESSING)

NAMING

si può risolvere attraverso un numero associato staticamente alla interfaccia del servizio

ADDRESSING DINAMICO

1) si può risolvere con un multicast o broadcast attendendo solo la prima risposta e non le altre

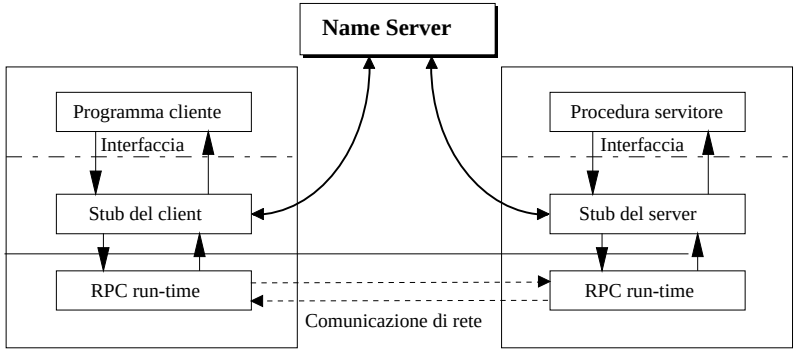
ESPLICITAMENTE dai processi

2) uso di un ***name server*** che registra tutti i servitori

IMPLICITAMENTE dall'agente esterno

Azioni sulle tabelle di binding

registrazione, aggiornamento, eliminazione



Binding Dinamico

La chiamata può andare a buon fine dopo un collegamento statico o dinamico

il binding può avvenire meno frequentemente delle chiamate stesse

in genere, ***si usa lo stesso binding per molte richieste e chiamate allo stesso server***

Binder (Broker, Name Server, etc.)

entità che consente il collegamento

tra **clienti e servitori**

possibilità di tenere conto dei **servizi**

operazioni per un binder
possono consentire anche agganci più liberi

lookup (servizio, versione, &servitore)
register (servizio, versione, servitore)
unregister (servizio, versione, servitore)

Il **nome del servitore** (servitore) può essere dipendente dal nodo di residenza o meno: se dipendente, allora una variazione deve essere comunicata al binder

BINDING come servizio coordinato

Uso di **binder multipli** per limitare overhead

Inizialmente i clienti usano un broadcast per trovare il binder più conveniente

Uso di **cache** ai singoli clienti o ai singoli nodi

GESTORE del Binding Dinamico

La chiamata sono, almeno in generale, dinamiche e richiedono un gestore opportuno

Presenza di gestori

Si possono ipotizzare gestori con politiche molto diverse

Unico gestore centrale
Più **gestori isolati** (ognuno gestisce una partizione)
Più **gestori coordinati**

A secondo del costo che si vuole sopportare

La maggior parte dei sistemi di RPC e derivati tende ad ipotizzare

NON un unico gestore centralizzato
MA più gestori partizionati e non coordinati

Allocazione del gestore

In caso di **gestori molteplici**,
ogni gestore tende ad essere responsabile
di una area di gestione

Tipicamente il gestore risiede su un nodo e gestisce le operazioni di quel nodo
I server del nodo si registrano presso il gestore locale
I clienti devono accedere al gestore prima di arrivare al servizio sul nodo stesso

JAVA RMI
(Remote Method Invocation)

RPC IN JAVA

Le RMI introducono la possibilità di richiedere le esecuzione di metodi remoti in JAVA integrando il tutto con il paradigma OO

Definizioni e generalità

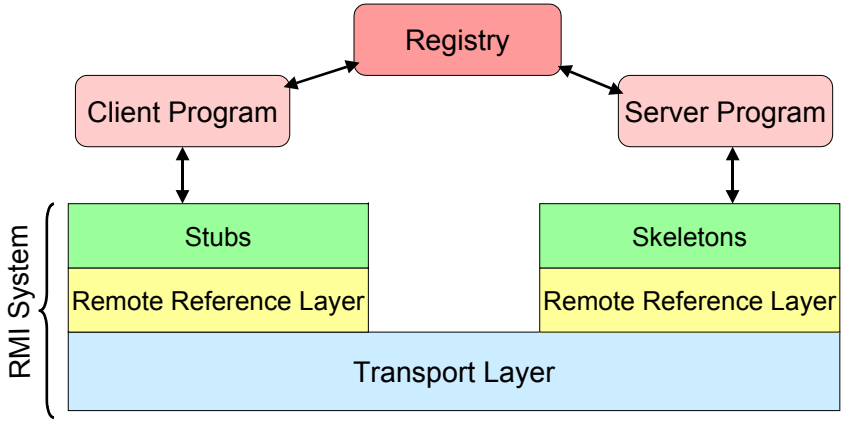
RMI
Insieme di politiche e meccanismi che permettono ad un'applicazione Java in esecuzione su una macchina di invocare i metodi
di un'altra applicazione Java in esecuzione su una macchina remota

> **Viene creato localmente solo il** riferimento ad un oggetto remoto**, che è effettivamente attivo su un host distinto. Un programma invoca i metodi attraverso questo riferimento locale.**

Si riescono quindi a superare tutti i problemi di eterogeneità di ambienti grazie alla omogeneità del supporto al linguaggio Java

Dobbiamo sempre considerare i sistema dei nomi che consente il legame dinamico tra clienti e servitore

Architettura RMI



Stub: **proxy locale su cui vengono fatte le invocazioni destinate all'oggetto remoto**
Skeleton: **elemento remoto che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sul server (ormai solo per back compatibility)**

Remote Reference Layer:
- fornisce il supporto alle chiamate inoltrate dallo stub
- definisce e supporta la semantica dell'invocazione e della comunicazione

Transport Layer: localizza il server RMI relativo all'oggetto remoto richiesto, gestisce le connessioni (TCP/IP, timeout) e le trasmissioni (sequenziali, serializzate), usando ***Java Remote Method Protocol (JRMP).***

Registry: servizio di naming che consente al server di pubblicare un servizio e al client di recuperarne il proxy

Caratteristiche

Modello a oggetti distribuito

Nel modello ad oggetti distribuito di Java un *oggetto remoto* consiste in:

- un oggetto con **metodi invocabili** da un'altra JVM, in esecuzione su un host differente

- un oggetto descritto **tramite interfacce remote** che dichiarano i metodi accessibili da remoto

Obbiettivo
Rendere il più possibile simili la chiamata locale vs. chiamata remota

SINTASSI: UGUALE => TRASPARENZA
Semantica: diversa semantica

Chiamate locali:
affidabilità ≈ 100%
funziona tutto o si guasta tutto

Chiamate remote:
possibilità di fallimento
di solamente una delle due macchine

INTERFACCE E IMPLEMENTAZIONE

Separazione tra

definizione del comportamento => interfacce

implementazione del comportamento => classi

Per realizzare componenti utilizzabili in remoto:

1. definire comportamento -> interfaccia che
   - estende **`java.rmi.Remote`** e
   - propaga **`java.rmi.RemoteException`**

2. implementare comportamento -> classe che
   - implementa l'interfaccia
   - estende **`java.rmi.UnicastRemoteObject`**

Il **server** deve quindi:
derivare dalla classe **`UnicastRemoteObject`**
e implementare la interfaccia remota stabilita

la interfaccia del server deve
derivare dalla interfaccia **`Remote`** a sua volta

Uso di RMI

Come **strumenti di supporto** ad RMI:
sia l'usuale compilatore Java (o ambienti)
sia un compilatore di Interfacce che produce stub e skeleton (**RMI compiler o rmic**)

Lo sviluppo di un'applicazione distribuita usando RMI:

1. Definire interfacce e generare stub e skeleton delle classi utilizzabili in remoto (con **rmic**)

2. Progettare le implementazioni dei componenti utilizzabili in remoto e compilare le classi necessarie (con **javac**)

3. Pubblicare il servizio
   - attivare il registry
   - registrare il servizio
   (il **server** deve fare una **bind sul registry**)

4. Il **client** deve ottenere il reference all'oggetto remoto facendo un **lookup sul registry**

A questo punto l'interazione tra il client e il server può procedere

N.B.: questa è una descrizione di base, dettagli ulteriori seguono (sul registry e sul caricamento dinamico delle classi)

Esempio: servizio di echo

Definizione dell'interfaccia del servizio
L'interfaccia deve

- **ereditare dalla interfaccia Remote**
- **tenere conto delle eccezioni da Remote**

```
public interface EchoInterface
                        extends   java.rmi.Remote
{String  getEcho(String echo)
  throws java.rmi.RemoteException;
}
```

L'interfaccia del servizio è il contratto che deve essere rispettato dai servitori e anche dai cliente a cui deve essere noto

Implementazione del Server

```
public class EchoRMIServer
extends java.rmi.server.UnicastRemoteObject
implements EchoInterface
{
// deve ereditare da UniCastRemote e
// implementare la interfaccia definita

// Costruttore
public EchoRMIServer()
throws java.rmi.RemoteException
  { super(); }


// Implementazione del metodo in interfaccia
public String getEcho(String echo)
throws java.rmi.RemoteException
  { return echo; }


public static void main(String[] args)
{
try{
EchoRMIServer serverRMI = new EchoRMIServer();
Naming.rebind("EchoService", serverRMI);
// Registrazione del servizio
   }
catch (Exception e)
  {e.printStackTrace(); System.exit(1); }
 }
}
```

Implementazione del Client

```
public class EchoRMIClient
{

// Avvio del Client RMI
public static void main(String[] args)
{BufferedReader stdIn= new BufferedReader
        (new InputStreamReader(System.in));
 try
  {

// Connessione al registry RMI remoto
// si deve ricercare la interfaccia e
// ritiparla in modo corretto
 EchoInterface serverRMI = (EchoInterface)
    java.rmi.Naming.lookup("EchoService");

// Interazione con l'utente
 String message, echo;
 System.out.print("Messaggio? ");
 message = stdIn.readLine();


// Richiesta del servizio remoto
 echo = serverRMI.getEcho(message);
 System.out.println("Echo: "+echo+"\n");
  }

 catch (Exception e)
  {e.printStackTrace(); System.exit(1);}
 }
}
```


PASSI di SVILUPPO

Creazione dello Stub e dello Skeleton

Con il compilatore RMI:
rmic EchoRMIServer

si generano i file dello stub e dello skeleton
EchoRMIServer_Stub.class
EchoRMIServer_Skel.class

Compilazione

javac EchoInterface.java
EchoRMIClient.java
EchoRMIServer.java

Esecuzione

1) Avviamento del registry: **rmiregistry**

2) Avviamento del server:
java EchoRMIServer

3) Avviamento del client:
java EchoRMIClient

La serializzazione

Marshalling: processo di codifica di argomenti e risultati per la trasmissione.
Unmarshalling: processo inverso di decodifica di argomenti e risultati ricevuti.

In Java questo problema è risolto usando la *serializzazione*, che viene fatta in maniera trasparente dal supporto

Serializzazione: trasformazione di oggetti complessi in semplici sequenze di byte

=> metodo **`writeObject()`**
su uno stream di output

Deserializzazione: decodifica di una sequenza di byte e costruzione di una copia dell'oggetto originale

=> metodo **`readObject()`**
da uno stream di input

In Java sono utilizzati in molti casi:
- per memoria anche esterna
- trasmissione tra macchine diverse
  (parametri e valori di ritorno in RMI)

Esempio di storage

```
Record record = new Record();
FileOutputStream fos = new  FileOutputStream("data.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(record);
```

```
FileInputStream fis = new FileInputStream("data.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
record = (Record)ois.readObject();
```

Si possono serializzare soltanto istanze di oggetti serializzabili, ovvero che:

- **implementano l'interfaccia `Serializable`**
- contengono **esclusivamente oggetti** (o riferimenti a oggetti) **serializzabili**

NOTA BENE:

NON viene trasferito l'oggetto vero e proprio ma solo le informazioni che ne caratterizzano l'istanza
=> no metodi, no costanti,
no variabili **`static`**, no variabili **`transient`**

Al momento della deserializzazione sarà **ricreata una copia** dell'istanza "trasmessa" usando il .class (che deve quindi essere accessibile!!!) dell'oggetto e le informazioni ricevute.

Passaggio di parametri

Tipo	Metodo Locale	Metodo Remoto
Tipi primitivi	Per valore	Per valore
Oggetti	Per riferimento	Per valore (deep copy)
Oggetti Remoti Esportati	Per riferimento	Per riferimento remoto

Shallow Copy vs. Deep Copy
considerando un grafo di oggetti

Passaggio per valore => Serializable Objects
Passaggio per riferimento => Remote Objects

Oggetti serializzabili

Oggetti la cui locazione non è rilevante per lo stato
Sono passati **per valore**: ne viene serializzata l'istanza che sarà deserializzata a destinazione per crearne una copia locale.

Oggetti remoti

Oggetti la cui funzione è strettamente legata alla località in cui eseguono (server)
Sono passati **per riferimento**: ne viene serializzato lo stub, creato automaticamente dal proxy (stub o skeleton) su cui viene fatta la chiamata in cui compaiono come parametri

Esempio

Riprendendo il server di echo => messaggio come oggetto anziché come stringa

```
public class Message
    implements Serializable
{

  String content;

   // … altri eventuali campi

  public Message(String msg)
  {
   content=msg;
  }

  public String toString()
  {
   return content;
  }

}
```

Quindi potremmo sostituire alla stringa una istanza della classe Message

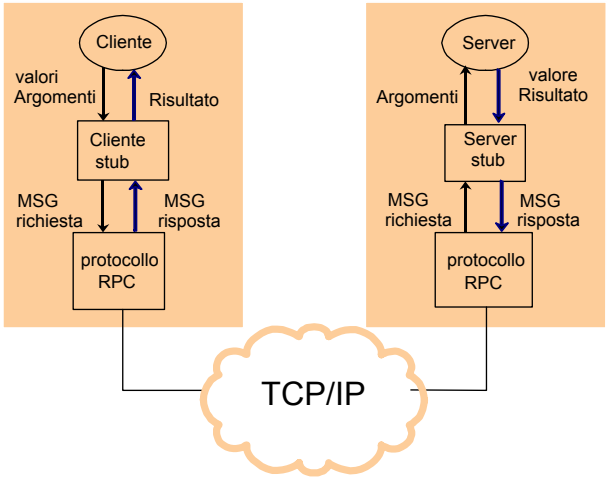
Gli STUB

Quali azioni devono fare?

stub cliente:
<ricerca del **servitore**>
<***marshalling*** argomenti>
<send richiesta>
<receive risposta>
<***unmarshalling*** risultato>
restituisci risultato
fine stub cliente;

stub servitore:
<attesa della richiesta>
<***unmarshalling*** argomenti>
invoca operazione locale
ottieni risultato
<***marshalling*** del risultato>
<send risultato>
fine stub servitore;

In aggiunta, il controllo della **operazione** ed eventuali azioni di **ritrasmissione** (entro un tempo di durata massima)



Stub e Skeleton

Stub e Skeleton sono oggetti generati dal compilatore RMI che gestiscono

marshalling/unmarshalling e
comunicazione (socket) tra client e server

Procedura di comunicazione:

1. il client ottiene un'istanza dello stub
2. il client chiama metodi sullo stub
3. lo stub:
   - crea una connessione con lo skeleton (o ne usa una già esistente)
   - marshalling delle informazioni associate alla chiamata (id del metodo e argomenti)
   - invia le informazioni allo skeleton
4. lo skeleton:
   - unmarshalling dei dati ricevuti
   - effettua la chiamata sull'oggetto che implementa il server
   - marshalling del valore di ritorno e invio allo stub
5. lo stub:
   - unmarshalling del valore di ritorno e restituzione del risultato al client

Stub

Estende java.rmi.server.RemoteStub
Implementa java.rmi.Remote
InterfacciaRemotaServer
(es. EchoInterface)

Realizza le chiamate ai metodi dell'interfaccia remota del server usando il **metodo `invoke`(…)** di **`java.rmi.server.RemoteRef`**

```
public final class EchoRMIServer_Stub
extends java.rmi.server.RemoteStub
implements EchoInterface, java.rmi.Remote
{
    …
// implementazione di getEcho
 public Message getEcho(Message message)
        throws java.rmi.RemoteException
 {
 try
 {              …
  // creazione della chiamata
  java.rmi.server.RemoteCall remotecall =
     super.ref.newCall(this, operations, 0,
     0xca41fff3e3260b7aL);
  // serializzazione dei parametri
   try
   { ObjectOutput objectoutput =
         remotecall.getOutputStream();
          objectoutput.writeObject(message);
   }
// cattura eccezioni di marshalling
          …

// invio della chiamata
  super.ref.invoke(remotecall);
```



```
// deserializzazione del valore di ritorno
   Message message1;
   try
   {
    ObjectInput objectinput =
        remotecall.getInputStream();
    message1 = (Message)objectinput.readObject();
    }

// cattura eccezioni di unmarshalling
          …

// segnalazione chiamata andata a buon fine

   finally
     {super.ref.done(remotecall);}

// restituzione del risultato
   return message1;
        }

// cattura eccezioni residue
          …
    }

    …
}
```

Skeleton

Implementa java.rmi.server.Skeleton

Inoltra le richieste al server usando il **metodo `dispatch(…)`**

```
public final class EchoRMIServer_Skel
     implements Skeleton
{    …
 public void dispatch(Remote remote,
   RemoteCall remotecall, int opnum, long hash)
     throws Exception
  {
// validazione e verifica di errori
      …
EchoRMIServer echormiserver =
                (EchoRMIServer)remote;
switch(opnum)
{case 0: // '\0'
   Message message;
 try
   {
// deserializzazione dei parametri di invocazione
    ObjectInput objectinput =
                 remotecall.getInputStream();
    message = (Message)objectinput.readObject();
   }
   catch(IOException ioexception1)
   {throw new UnmarshalException("error
      unmarshalling arguments", ioexception1);
   }
   catch(ClassNotFoundException clexc)
   {throw new UnmarshalException("error
      unmarshalling arguments", clexc);
   }
   finally
         {remotecall.releaseInputStream();}
```


Parte di invocazione

```
 // effettiva invocazione del metodo sul server
Message message1 =
      echormiserver.getEcho(message);
 try
  {
 // serializzazione del valore di ritorno
  ObjectOutput objectoutput =
        remotecall.getResultStream(true);
 objectoutput.writeObject(message1);
  }

  catch(IOException ioexception)
  { throw new MarshalException("error
     marshalling return", ioexception);
  }
  break;

  default:
    throw new UnmarshalException
                ("invalid method number");
  }


  }

    …
}
```

Interazione Client/Server
Socket e Thread

Client:

L'invocazione di un metodo remoto implica la connessione (Socket) con l'oggetto remoto

Condivisione delle connessioni tra la JVM client e la JVM server: richiesta di connessione per una stessa JVM

- se c'è una connessione libera aperta => riuso
- se non c'è una connessione libera aperta => nuova connessione

Al completamento di una operazione, la connessione viene liberata e rimane attiva fino allo scadere di un timeout

Server:

L'esportazione di un oggetto remoto provoca la creazione di

- una socket d'ascolto (**Server Socket**)
- un thread che riceve le richieste (**Listener Thread**)

associati all'oggetto remoto

Uso delle risorse

Condivisione delle server socket e dei listener: server in esecuzione nella stessa JVM possono condividere la porta d'ascolto (soluzione di default se non la specificano) e il listener che ne gestisce le richieste

L'accettazione di una richiesta provoca la **creazione di un thread** che esegue la richiesta (**Server Thread**)

Condivisione dei server thread: allocazione di un thread per ogni connessione con un client, riutilizzato in maniera analoga alla connessione di cui esegue la richiesta

RMI permette a più thread di accedere ad uno stesso oggetto server (**gestione concorrente delle richieste**)

=> l'implementazione dei metodi remoti deve essere *thread-safe*

necessità di garantire la mutua esclusione quando è il caso

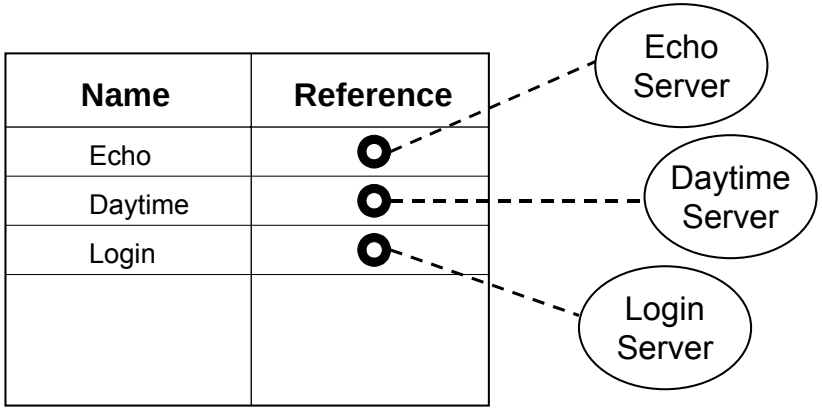
RMI Registry

Problema di ***Bootstrapping***: un client in esecuzione su un macchina ha bisogno di localizzare un server a cui vuole connettersi, che è in esecuzione su un'altra macchina. Tre possibili soluzioni:

- Il client conosce in anticipo dov'è il server
- L'utente dice all'applicazone client dov'è il server (es. e-mail client)
- Un servizio standard (***naming service***) in una locazione ben nota, che il client conosce, funziona come *punto di indirezione*

Java RMI utilizza un naming service: ***RMI Registry***

Mantiene un insieme di coppie **{name, reference}**
Name: stringa arbitraria non interpretata



Operazioni

Metodi della classe ***java.rmi.Naming***:

```
public static void bind(String name, Remote obj)
public static void rebind(String name, Remote obj)
public static void unbind(String name)
public static String[] list(String name)
public static Remote lookup(String name)
```

name -> combina la locazione del registry e il nome logico del servizio, nel formato:

//registryHost:port/logical_name

A default:
registryHost = macchina su cui esegue il programma che invoca il metodo
port = 1099

Il Registry è un server RMI
in esecuzione su registryHost
in ascolto su port

Ognuno di questi metodi crea una connessione (socket) con il registry identificato da host e porta

Implementazione del Registry

Il Registry è un server RMI

Classe d'implementazione:
sun.rmi.registry.RegistryImpl

Interfaccia: *java.rmi.registry.Registry*

```
public interface Registry extends Remote {

    public static final int REGISTRY_PORT = 1099;

    public Remote lookup(String name)
        throws RemoteException, NotBoundException,
                            AccessException;
    public void bind(String name, Remote obj)
        throws RemoteException, AlreadyBoundException,
                            AccessException;
    public static void rebind(String name, Remote obj)
             throws  RemoteException, AccessException;
    public static void unbind(String name)
    throws RemoteException, NotBoundException,
                            AccessException;
    public static String[] list(String name)
        throws  RemoteException, AccessException;
}
```

Perchè usare anche la classe Naming?

Distribuzione delle classi

In una applicazione RMI è necessario che siano disponibili gli opportuni file .class nelle località dove necessario (per l'esecuzione o per la deserializzazione)

Il Server deve poter accedere a:
- interfacce che definiscono il servizio
- implementazione del servizio
- stub e skeleton delle classi di implementazione
- altre classi utilizzate dal server

Il Client deve poter accedere a:
- interfacce che definiscono il servizio
- stub delle classi di implementazione del servizio
- classi del server usati dal client (es. valori di ritorno)
- altre classi utilizzate dal client

È necessario:
1. localizzare il codice (in locale o in remoto)
2. effettuare il download (se in remoto)
3. eseguire in modo sicuro il codice scaricato

Localizzazione del codice

Le informazioni relative a dove reperire il codice sono memorizzate sul server e sono passate al client by need

⇒ server RMI mandato in esecuzione specificando nell'opzione

`java.rmi.server.codebase`

l'URL da cui prelevare le classi necessarie.

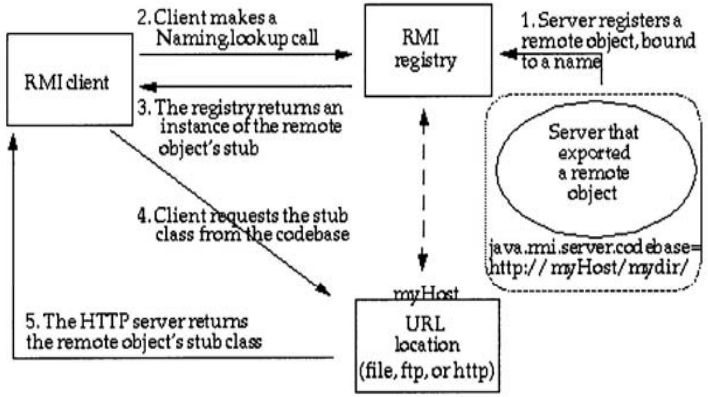
L'URL può essere:
una directory del file system locale (**file://**)
l'indirizzo di un server ftp (**ftp://**)
l'indirizzo di un server http (**http://**)

Il codebase è una proprietà del server che viene annotata nel reference pubblicato sul registry

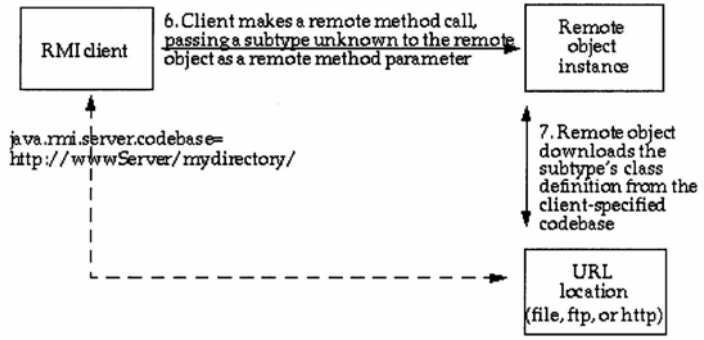
Le classi vengono cercate sempre **prima nel CLASSPATH locale**, solo in caso di insuccesso vengono cercate dove specificato dal codebase

Utilizzo del codebase

Il codebase viene usato dal **client** per scaricare le classi necessarie del server (interfaccia, stub, oggetti restituiti come valori di ritorno)



Il codebase viene usato dal **server** per scaricare le classi necessarie relative al client (oggetti passati come parametri nelle chiamate)



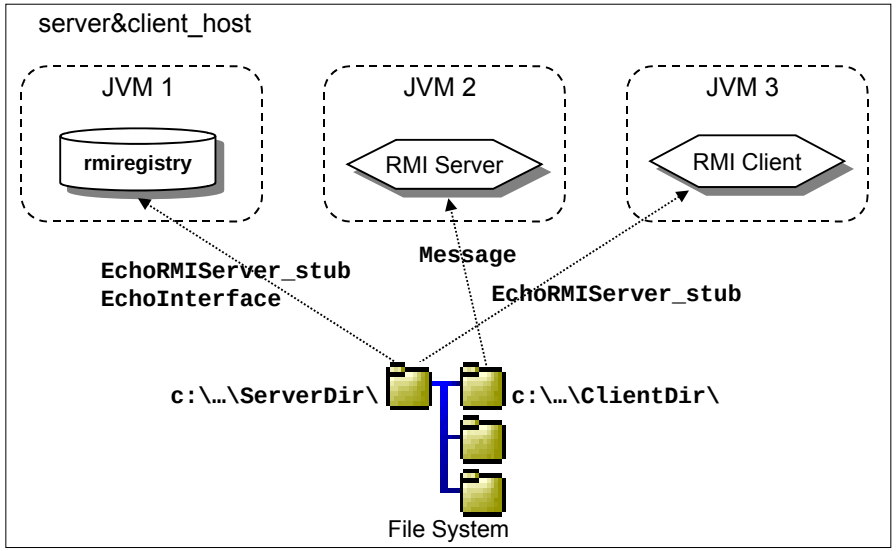
Uso del CODEBASE

Sia il server che il client devono essere lanciati specificando il codebase
per indicare dove sono disponibili le classi

Esempio con client e server sullo stesso host e classi in direttori diversi:

```
java    -Djava.rmi.server.codebase
                = file://c:\…\RMIdir\ServerDir\
        EchoRMIServer

java    -Djava.rmi.server.codebase
                = file://c:\…\RMIdir\ClientDir\
        EchoRMIClient localhost
```

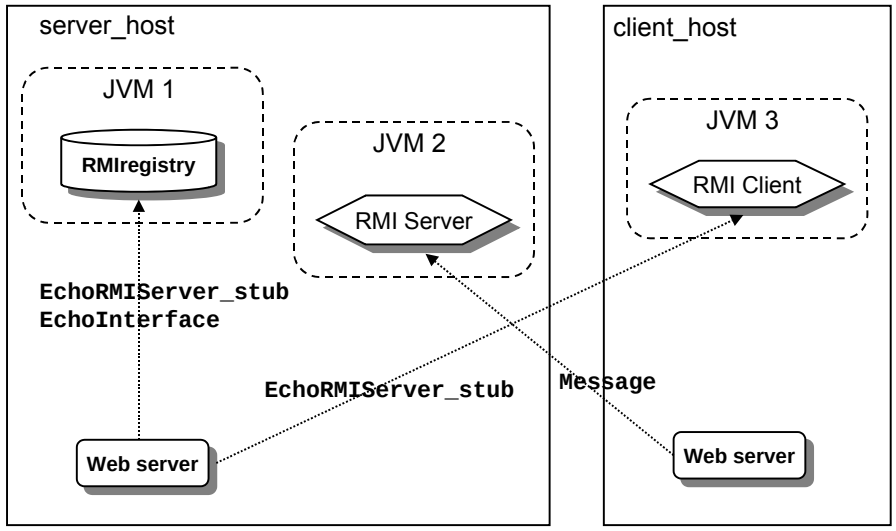


ESEMPIO

client e server su host diversi e classi scaricabili via http:

```
java
 -Djava.rmi.server.codebase
     = http://server_host_name/…/RMIdir/ServerDir
        EchoRMIServer

java
 -Djava.rmi.server.codebase
     = http://client_host_name/…/RMIdir/ClientDir
        EchoRMIClient server_host_name
```



DOWNLOAD del CODICE

Utilizzo di **`RMIClassLoader`**

usato dal supporto RMI per caricare le classi

Esecuzione del codice

Utilizzo di **`RMISecurityManager`**

Istanziato all'interno dell'applicazione RMI, se non ce n'è già uno istanziato, sia client che server

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

Sia il server che il client devono essere lanciati specificando il file con le autorizzazioni che il security manager deve caricare

Esempio

```
java -Djava.security.policy = echo.policy
      EchoRMIServer

java -Djava.security.policy = echo.policy
      EchoRMIClient remoteHost
```


File di Policy

Insieme dei permessi da garantire alle richieste esterne

Struttura del file delle policy:

```
grant
{

 permission java.net.SocketPermission
    "*:1024-65535", "connect, accept";

 permission java.net.SocketPermission
    "*:80", "connect";

 permission java.io.File Permission
  "c:\\home\\RMIdir\\-", "read";

};
```

Il primo permesso consente al client e al server di instaurare le connessioni necessarie all'interazione remota

Il secondo e il terzo permesso consentono di prelevare il codice rispettivamente da un server http e da una directory del file system locale

Garbage collection di oggetti remoti

In un sistema distribuito si vuole avere la deallocazione automatica degli oggetti remoti che non hanno più nessun riferimento presso dei client.

- **Il sistema RMI utilizza un algoritmo di garbage collection basato sul conteggio dei riferimenti**
- **Ogni JVM aggiorna una serie di *contatori* ciascuno associato ad un determinato oggetto.**
- **Ogni contatore rappresenta il numero dei riferimenti ad un certo oggetto che in quel momento sono attivi su una JVM.**
- **Ogni volta che viene creato un riferimento ad un oggetto remoto il relativo contatore viene incrementato. Per la prima occorrenza viene inviato un messaggio che avverte l'host del nuovo client.**
- **Quando un riferimento viene eliminato il relativo contatore viene decrementato. Se si tratta dell'ultima occorrenza un messaggio avverte il server.**
- **Quando nessun client ha più riferimenti ad un oggetto, il runtime di RMI utilizza un "*weak reference*" per indirizzarlo.**

Il *weak reference* è usato nel garbage collector per eliminare l'oggetto nel momento in cui anche dei riferimenti locali non sono più presenti.

Sistemi di RPC

proprietà visibili all'utilizzatore

entità che si possono richiedere
operazioni o metodi di oggetti

semantica di comunicazione
maybe, at most once, at least once

modi di comunicazione
sincroni, asincroni, sincroni non bloccanti

durata massima e eccezioni
ritrasmissioni e casi di errore

proprietà trasparenti all'utilizzatore

ricerca del servitore
uso di sistemi di nomi
broker unico centralizzato / broker multipli
su vari nodi

presentazione dei dati
linguaggio IDL ad hoc e generazione stub

passaggio dei parametri
passaggio per valore, per riferimento
integrazione con i servizi locali

eventuali legami con le risorse del server
persistenza della memoria per le RPC
crescita delle operazioni via movimento di codice
aggancio con il garbage collector

RPC di SUN

visione a processi e
non trasparenza alla allocazione
operazioni richieste al nodo del servitore

entità che si possono richiedere
operazioni
semantica di comunicazione
durata massima: at most once, at least once
modi di comunicazione
sincroni, asincroni
ricerca del servitore
port mapper sul server
presentazione dei dati
linguaggio IDL ad hoc XDR
e generazione stub RPCGEN
passaggio dei parametri
passaggio per valore
le strutture complesse e definite dall'utente sono linearizzate e ricostruite al server per essere distrutte al termine della operazione
estensioni varie
broadcast
credenziali per sicurezza
...
operazioni

RMI in Java

visione per sistemi ad oggetti passivi
trasparenza alla allocazione
non si prevede eterogeneità
scelte che non privilegiano la efficienza

entità che si possono richiedere
metodi di oggetti via interfacce remotizzabili
semantica di comunicazione
at most once (TCP)
se altro protocllo anche meno garanzie
modi di comunicazione
sincroni
durata massima e eccezioni
trattamento di casi di errore tramite il linguaggio
ricerca del servitore
uso di *registry* nel sistema
broker unico centralizzato / broker multipli
presentazione dei dati
generazione stub e skeleton intermediari
passaggio dei parametri
passaggio a default per valore,
passaggio per riferimento di oggetti con interfacce remotizzabili
eventuali legami con le risorse del server
aggancio con il sistema di sicurezza
security manager
aggancio con il garbage collector