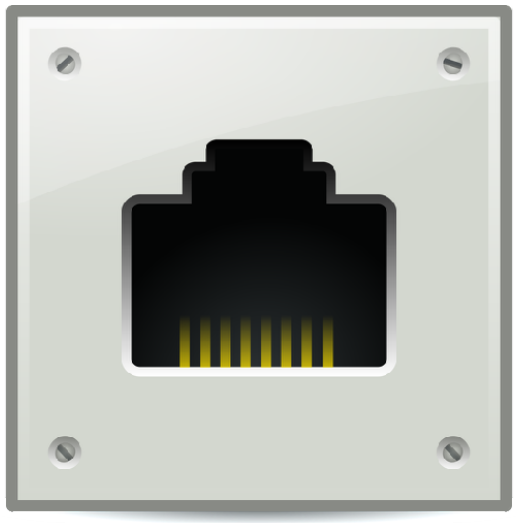
Java RMI



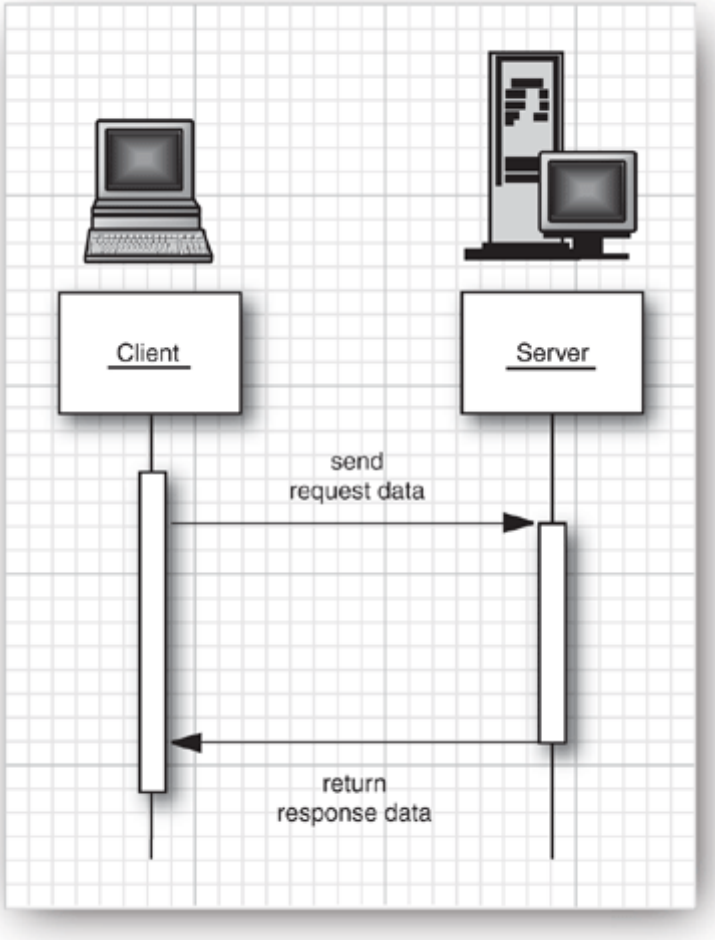
ing. Conti https://github.com/ingconti

Obiettivo

Poter allocare computazioni su nodi fisici diversi e consentire il coordinamento tra le computazioni sui diversi nodi

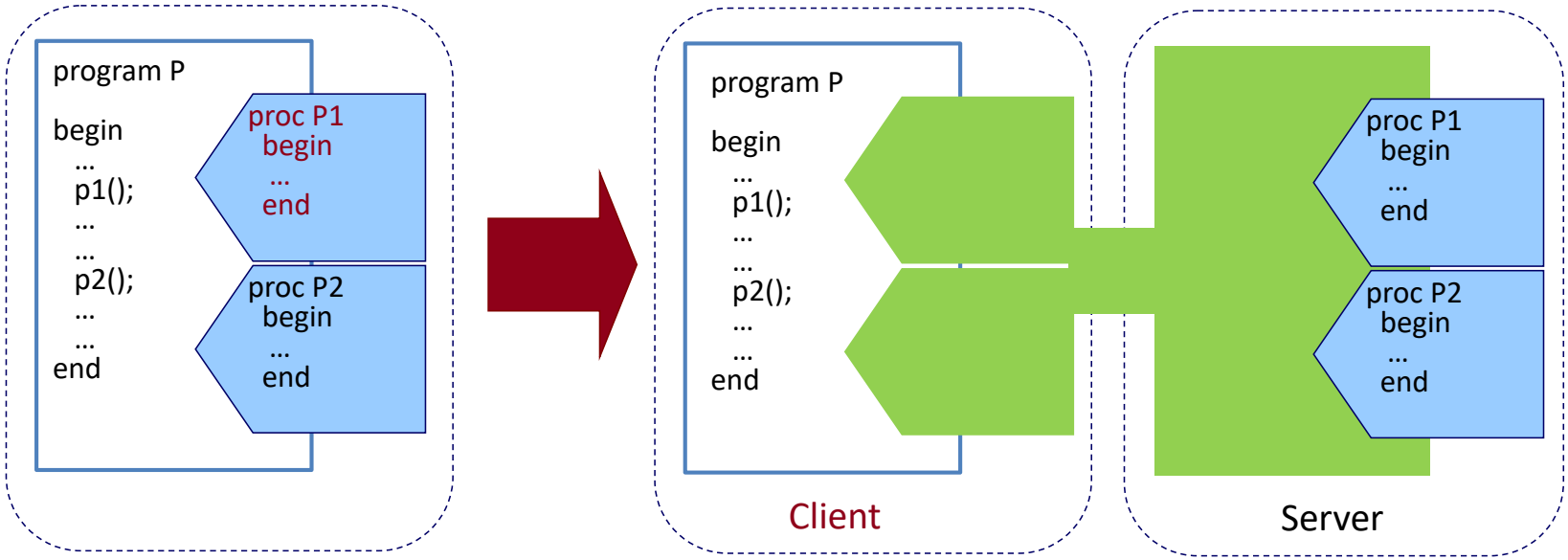
Verso RMI…

- L'idea alla base di tutta la programmazione distribuita è semplice:
  - Un client esegue una determinata richiesta
  - Tale richiesta viaggia lungo la rete verso un determinato server destinatario
  - Il server processa la richiesta e manda indietro la risposta al client per essere analizzata
  - Con i socket però dobbiamo gestire "a mano" il formato dei messaggi e la gestione della connessione



Cosa vorremmo?

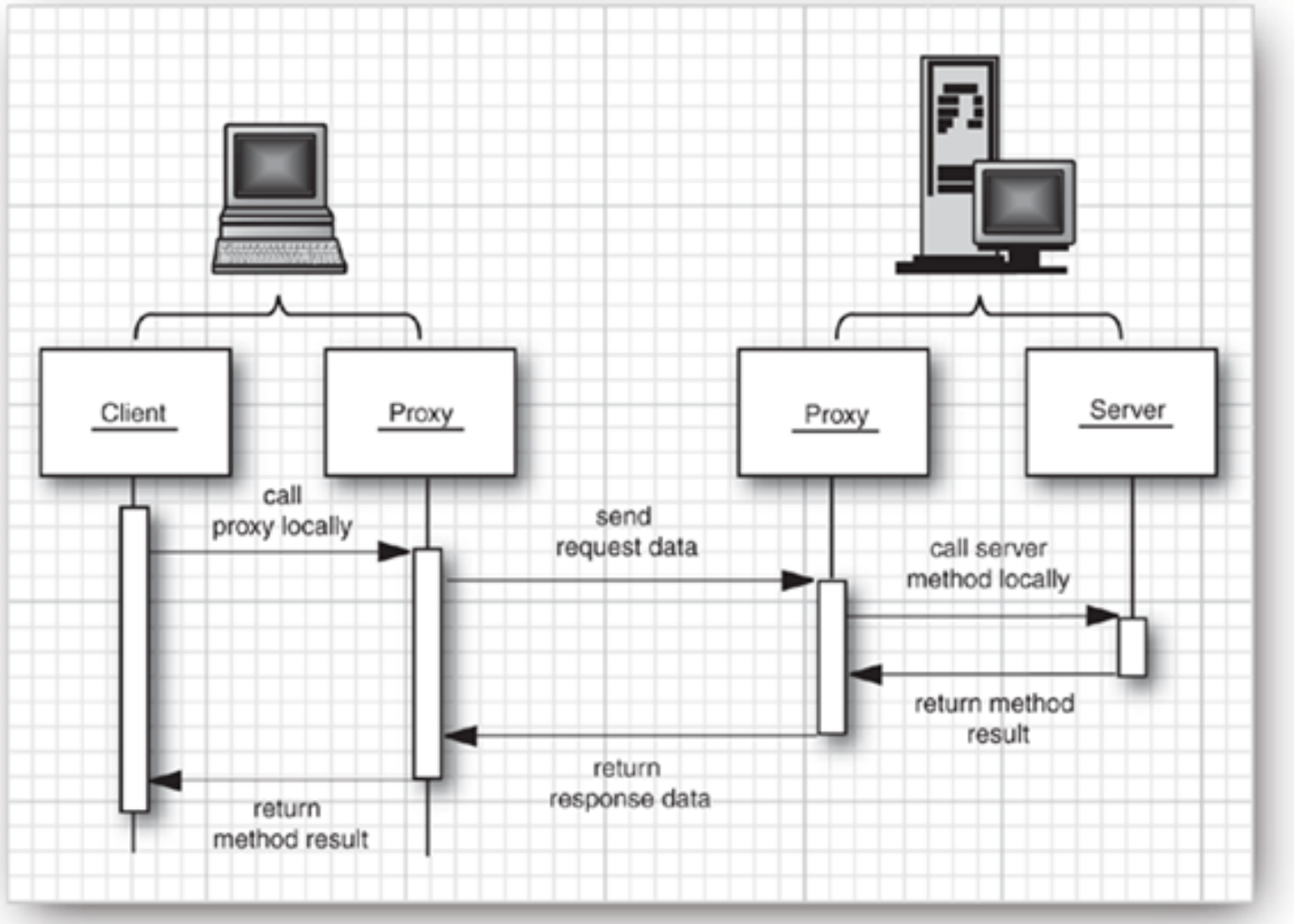
L'illusione che la rete non esistesse, e che le invocazioni a metodo funzionassero anche su oggetti "remoti"



Verso RMI

- Quello che cerchiamo è un meccanismo con il quale il programmatore del client esegue una normale chiamata a metodo
  - Senza preoccuparsi che c'è una rete di mezzo
  - Avremo 2 "intermediari:" ..
  - QUINDI 2 PROXY...

Verso RMI…


Client
Proxy
Proxy
Server
call
proxy locally
send
request data
call server
method locally
return method
result
return
response data
return
method result

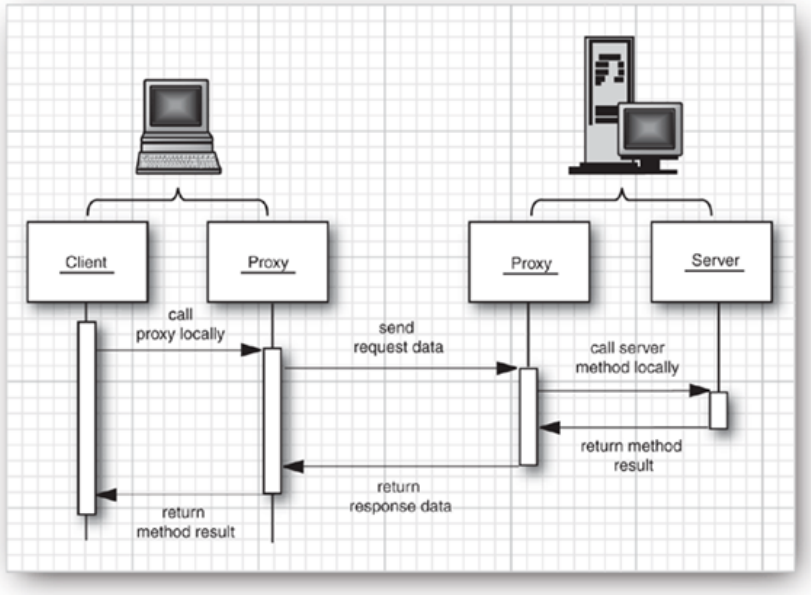
"ingredienti" di RMI

- Oggetto remoto
  - Oggetto i cui metodi possono essere invocati da una Java Virtual Machine diversa da quella dove l'oggetto risiede
- Interfaccia remota
  - Interfaccia che dichiara quali sono i metodi che possono essere invocati da una diversa Java Virtual Machine
- Server
  - Insieme di uno o più oggetti remoti che, implementando una o più interfacce remote, offrono delle risorse (dati e/o procedure) a macchine esterne distribuite sulla rete
- Remote Method Invocation (RMI)
  - Invocazione di un metodo presente in una interfaccia remota implementata da un oggetto remoto
  - La sintassi di una invocazione remota è identica a quella locale

Architettura interna

- Il client colloquia con un proxy locale del server, detto stub
  - Lo stub “rappresenta” il server sul lato client
  - Implementa l'interfaccia del server
  - È capace di fare forward di chiamate di metodi attraverso la rete

- Esiste anche un proxy del client sul lato server, detto skeleton
  - È una rappresentazione del client
  - Chiama i servizi del server
  - Sa come fare forward dei risultati attraverso la rete

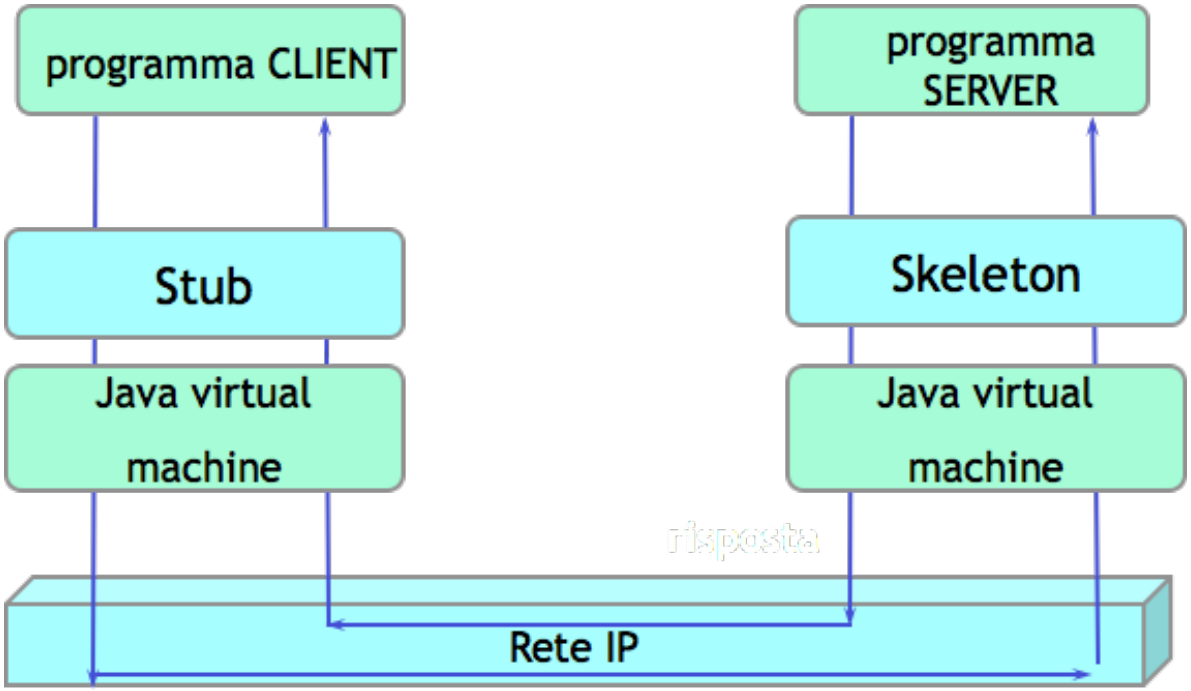


Documentazione ufficiale

- When a stub's method is invoked, it does the following:
  - Initiates a connection with the remote JVM containing the remote object,
  - Marshals (writes and transmits) the parameters to the remote JVM,
  - Waits for the result of the method invocation,
  - Unmarshals (reads) the return value or exception returned, and
  - Returns the value to the caller

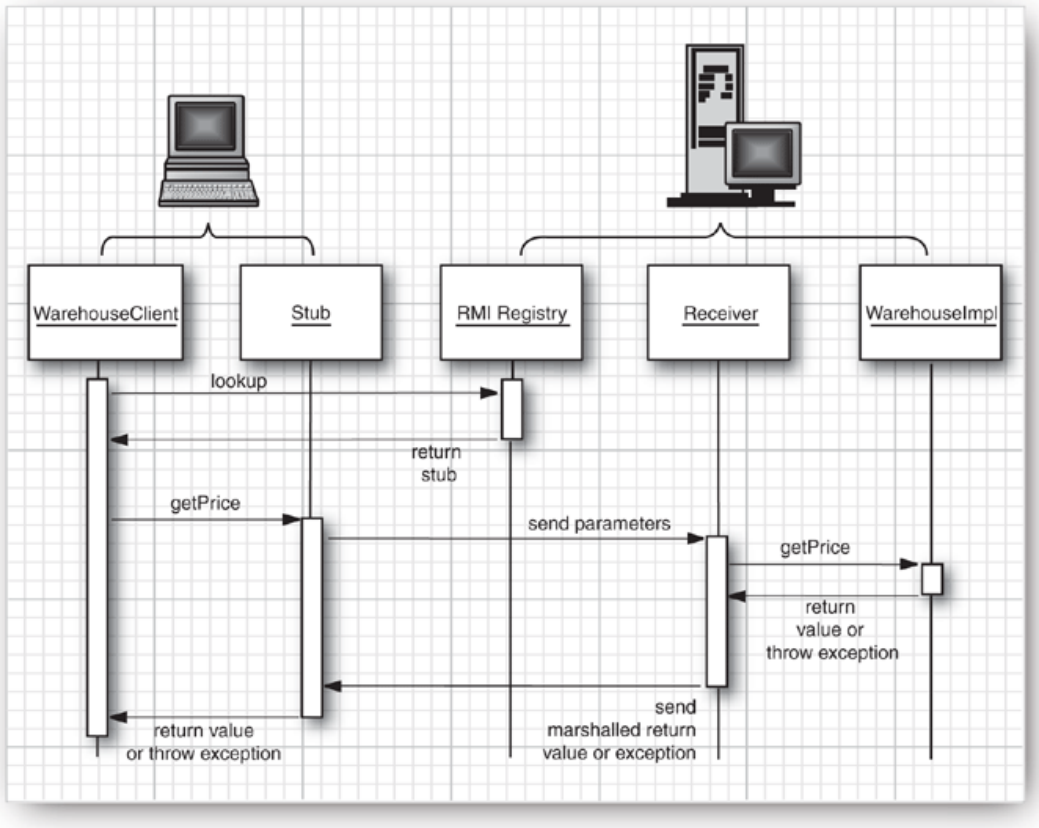
- Each remote object may have a corresponding skeleton:
  - Unmarshals (reads) the parameters for the remote method,
  - Invokes the method on the actual remote object implementation, and
  - Marshals (writes and transmits) the result (return value or exception) to the caller

Architettura interna



RMI Registry

- Il registro RMI si occupa di fornire al client lo stub richiesto
  - In fase di registrazione il server potrà fornire un nome canonico per il proprio oggetto remoto
  - Il client potrà quindi ottenere lo stub utilizzando il nome che gli è stato assegnato



Riassumendo…

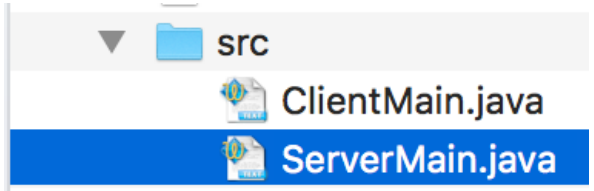
- Lato client
  - Viene richiesto a un registro RMI lo stub per l'invocazione di un determinato oggetto remoto
  - I parametri in ingresso all'invocazione remota vengono serializzati e trasmessi (marshalling)
  - L'invocazione remota viene inviata al server

- Lato server
  - Il server localizza l'oggetto remoto che deve essere invocato
  - Chiama il metodo desiderato passandogli i parametri ricevuti dal client
  - Cattura il valore di ritorno o le eventuali eccezioni
  - Spedisce allo stub del client un pacchetto contenente i dati ritornati dal metodo

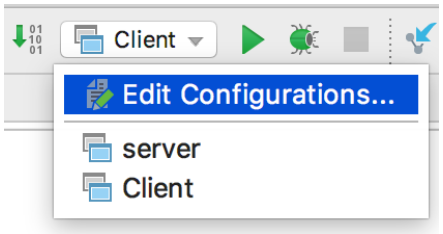
Prima due step banali:

creare un project con Main class: ClientMain
creare un project con Main class: ServerMain

ossia:



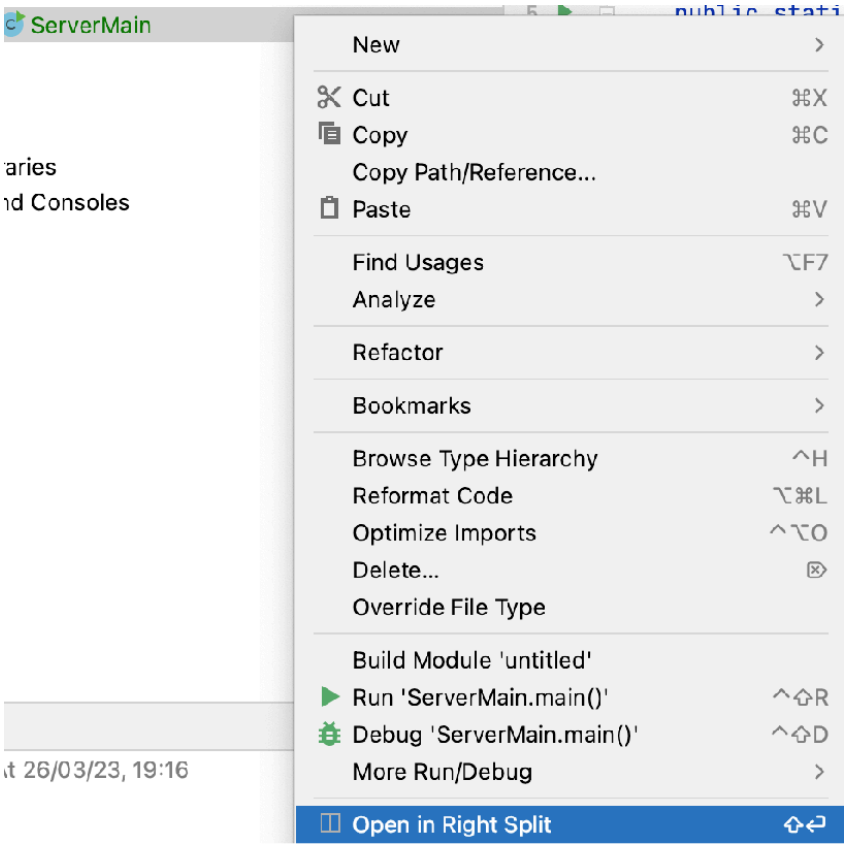
e i relativi 2 setup di test (intelliJ):



(Per le config. Basta un run... su ognuna)

proviamoli (RUN) e teniamo 2 finestre aperte: (split)

..



proviamoli (RUN) e teniamo 2 finestre aperte,

Run in debug..

```java
package org.example;

no usages  new *
public class ClientMain
{
    no usages  new *
    public static void main( String[] args )
    {
        System.out.println( "Hello from Client!" );
    }
}
```

```java
package org.example;

no usages  new *
public class ServerMain {

    no usages  new *
    public static void main( String[] args )
    {
        System.out.println( "Hello from Server!" );
    }
}
```

RMI todo list:

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program

(https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html

or:

https://www.tutorialspoint.com/java_rmi/java_rmi_application.htm
)

Define the remote interface *

perchè:

- va definita e va creato un OGGETTO che la implementi nel server
- NON si può farla implementare dal server stesso perchè va chiamata in un metodo **static**, mentre RMI vuole un oggetto:

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

OPPURE dovremmo far derivare il ns server da: UnicastRemoteObject

extends UnicastRemoteObject ..

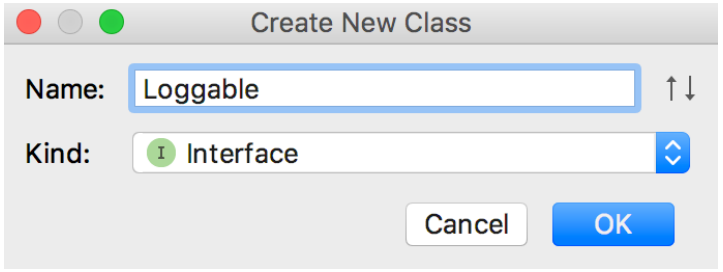
Scegliamo la 1'

*(code later..)

Define the remote interface

la interfaccia remota racchiude i metodi che il client cercherà ed seguirà sul server: (e che il server deve implementare)

p.es. **Loggable**



la interfaccia sarà sia nel Cl. che nel Srv.

Define the remote interface (2)

```
public interface Loggable  {

}
```

estende Remote quindi:

```
public interface Loggable extends Remote {
    boolean login(String nick) throws RemoteException;
     void    logout(String nick) throws RemoteException;
}
```

(al solito ALT Invio vi darà:

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

)
```

Develop the server program

(https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html)

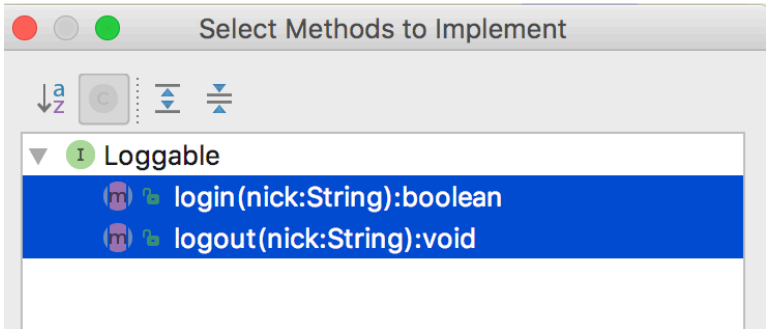
```
public class ServerMain{

    public static void main(String[] args) {
        System.out.println("Hello from Server!");
    }
}
```

Aggiungiamo:

```
public class ServerMain implements Loggable {
```

Develop the server program

chiederà:



e crea:

```
@Override
public boolean login(String nick) throws RemoteException {
    return false;
}

@Override
public void logout(String nick) throws RemoteException {

}
```

Develop the server program

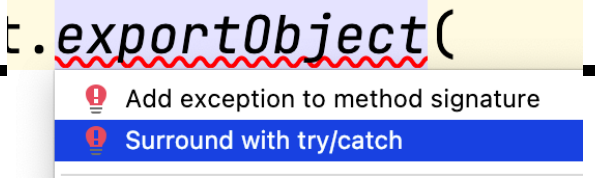
Il ns oggetto:

```
public class ServerMain implements Loggable {
    static int PORT = 1234;

    public static void main( String[] args )
    {
        System.out.println( "Hello from Server!" );
        ServerMain obj = new ServerMain();
       Loggable stub = (Loggable) UnicastRemoteObject.exportObject(
                obj, PORT);

}
```

Si noti che:

a) dobbiamo passare un oggetto della NS classe.

b) ***exportObject lancia eccezioni.. solite suggestions di IJ..***



Develop the server program

Il ns oggetto:

```
public class ServerMain implements Loggable {
    static int PORT = 1234;

    public static void main( String[] args )
    {
        System.out.println( "Hello from Server!" );
            ServerMain obj = new ServerMain();

try {
    Loggable stub = (Loggable) UnicastRemoteObject.exportObject(
                    obj, PORT);
} catch (RemoteException e) {
    e.printStackTrace();
}
```

..bind..

Develop the server program..bind

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

    ...
    try {
     Loggable stub = (Loggable)

    // Bind the remote object's stub in the registry
    Registry registry = LocateRegistry.createRegistry(PORT);
    registry.bind("Loggable", stub);
    System.err.println("Server ready");
```

NOTA:
Registry registry = LocateRegistry.getRegistry();
NON funziona: rida' una istanza, ***se la trova***, del Registry, ma NON INIZIALIZZA il registry (Vedi esempio Oracle..)

Ancora eccezioni.. solito..

the server program (all)

oracle sample:

```
public class Server implements Hello {
    public Server() {}

    public String sayHello() {
        return "Hello, world!";
    }

    public static void main(String args[]) {
        try {
            Server obj = new Server();
            Hello stub = (Hello)
                UnicastRemoteObject.exportObject(obj, 0);

            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

IL NS....

OUR CODE:

```
public class ServerMain implements Loggable {
    static int PORT = 1234;

    public static void main( String[] args )
    {
        System.out.println( "Hello from Server!" );
        Loggable stub = null;

        ServerMain obj = new ServerMain();
        try {
            stub = (Loggable) UnicastRemoteObject.exportObject(
                        obj, PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }

        // Bind the remote object's stub in the registry
        Registry registry = null;
        try {
            registry = LocateRegistry.createRegistry(PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        try {
            registry.bind("Loggable", stub);
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (AlreadyBoundException e) {
            e.printStackTrace();
        }
        System.err.println("Server ready");

    }

    @Override
    public boolean login(String nick) throws RemoteException {
        return false;
    }

    @Override
    public void logout(String nick) throws RemoteException {

    }
}
```

Dev. the server program

```
run..
```

```
…lib/idea_rt.jar" ServerMain
Connected to the target VM, address: '127.0.0.1:56194', transport: 'socket'
Server ready
```

Develop the client program

```
public class ClientMain {

    static int PORT = 1234;

    public static void main(String[] args) {

        try {
            // Getting the registry
            Registry registry = LocateRegistry.getRegistry("127.0.0.1", PORT);
...
```

Il client prende instanza del registry, ma a diff. Del server, NON deve crearlo.

Ora come detto, invochiamo lo stub:

Develop the client program

```
public class ClientMain {

    static int PORT = 1234;

    public static void main(String[] args) {

        try {
            // Getting the registry
            Registry registry = LocateRegistry.getRegistry("127.0.0.1", PORT);

            // Looking up the registry for the remote object
           Loggable stub = (Loggable) registry.lookup("Loggable");

           // Calling the remote method using the obtained object
           Boolean logged = stub.login("Bob");

            System.out.println("Remote method invoked " + logged);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }

    }
}
```

AL Run da errore se server non è Up.

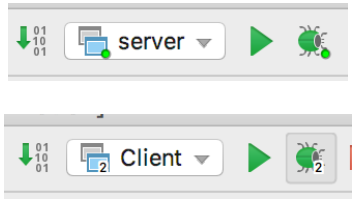
run together set breakpoints

```
public boolean login(String nick) throws RemoteException {
    System.out.println(nick +  " is logging..");
    return false;
}
```

```
// Calling the remote method using the ob
Boolean logged = stub.login( nick: "Bob");
```

1) run server (in debug)

2) run client (in debug)



run together set breakpoints (cont ‘d)

step by step del client.. si fermerà al breakpoint di Loggable

```
    public boolean login(String nick) throws RemoteException {  nick: "Bob"

        System.out.println(nick +  " is logging..");  nick: "Bob"
        return false;
    }
```

(per ora torna false..)

run together set breakpoints (cont ‘d)

```
Server ready
Bob is logging..
```

client:

```
Connected to the target VM,
address: '127.0.0.1:56311',
transport: 'socket'
Remote method invoked false
```

Da shell

Poiché in Java esiste la alberatura dei package,

(Nel ns caso .... org.example)

Per usare la cmd line, portarsi SULLA cartella ....main/java:

Quindi pes.

```
cd /Volumes/BigSur_110/Users/ingconti/Documents/GIT-BigSur/
insubria_pcd/2022/ES04/PROJECTS/rmi02/src/main/java
```

E qui:

```
javac org/example/ServerMain.java
```

```
javac org/example/ClientMain.java
```

Run (prima server):

```
java org/example/ServerMain
...
```

Da shell

Altra shell:

```
cd /Volumes/BigSur_110/Users/ingconti/Documents/GIT-BigSur/
insubria_pcd/2022/ES04/PROJECTS/rmi02/src/main/java
```

```
java org/example/ClientMain
```

Anche chiamato piu volte...

```
Remote method invoked false
```

Note:

- A differenza del server TCP, il server rimane sempre in attesa, anche senza i loop che abbiamo fatto

- RMI è pensato per mantenere viva la connessione lato server per garantire l'invocazione remoto dei client

- provate a lasciare aperta una SHELL con il server e a connettere disconnettere più volte il client…

Proviamo ad implementare un server
di e-commerce minimale

- Partiamo dal server precedente
- creiamo un class Product
- il server mandera' una lista prodotti alla richiesta del/dei client
- il client sceglierà Prodotto e la qta
- il metodo totale ci dara' il totale

Sia rm03

Another RMI sample…

- creiamo un class Product

```
public class Product {
    double cost;
    String descr;


    public Product(double cost, String descr) {
        this.cost = cost;
        this.descr = descr;
    }
}
```

Soliti costruttore / setter getter...toString..

creiamo un class Product ed un array nel server fisso:

(Dopo i vari metodi..)

```
ArrayList<Product> products = new ArrayList<>();
void buildProductList() {
    products.add(new Product(1, "apple"));
    System.out.println(products);
}
```

buildProductList nel main.. ma va creato costruttore
(Useremo metodi di istanza..)

Costruttore:

```
public class ServerMain implements Loggable {
    static int PORT = 1234;

    public ServerMain() {
        buildProductList();
    }

...
```

Classe CartRow:

.. e ripetiamo:

```
public class CartRow {

    Product product;
    int qty;


Constructor..

Il client ci dira indice prodotto e qta come detto, e li
metteremo nell' array:

ArrayList<CartRow> rows = new ArrayList<>();
```

Modifichiamo la Interface, da Loggable a "Sellable":

Un po' di refactor...

E nuovi metodi:

getList

buyProduct

getTotal

E qui:

```
// Looking up the registry for the remote object
Sellable stub = (Sellable) registry.lookup("Sellable");
ATTENZIONE! Sia server che client
(Si può anche tenere Logggable.. e' solo un nome... ma deve essere
uguale..)
```

un po' di refactor...

```
public interface Sellable extends Remote {
    ArrayList<Product> getList() throws RemoteException;
    double getTotal() throws RemoteException;
    double buyProduct(Product p, int qty) throws RemoteException;
}
```

... creiamoli..

un po' di refactor... anche sul CLIENT:

```
..
// Looking up the registry for the remote object
Sellable stub = (Sellable) registry.lookup("Sellable");

// Calling the remote method using the obtained object
ArrayList<Product> products = stub.getList();

System.out.println("Remote products " + products);
```

Ma sul server non ridiamo la lista...

Sul server:

```
@Override
public ArrayList<Product> getList() throws RemoteException
{
    return products;
}
```

Compile and run...

```
javac org/example/ServerMain.java
```

```
java org/example/ServerMain
```

Compile and run client ...

```
 javac org/example/ClientMain.java
```

```
java org/example/ClientMain
```

.....

Spara eccezioni..

..

```
java.io.NotSerializableException: org.example.Product
```

.....

Product deve implementare "Serializable":

```
public class Product implements Serializable {
```

Quindi nel codice lato server:

```
@Override
public ArrayList<Product> getList() throws RemoteException {
    return products;
}
....
```

Ora la arryList esce:

```
Remote products [Product{cost=1.0, descr='apple'}, Product{cost=2.0,
descr='orange'}]
```

Manca da implementare il totale... facciamo solo un caso giusto x prova.. senza I/O manuale:

Product deve implementare "Serializable":

```
public class Product implements Serializable {
```

Quindi nel codice lato server:

```
@Override
public ArrayList<Product> getList() throws RemoteException {
    return products;
}
....
```

Ora la arryList esce:

```
Remote products [Product{cost=1.0, descr='apple'}, Product{cost=2.0,
descr='orange'}]
```

Manca da implementare il totale... facciamo solo un caso giusto x prova..
senza I/O manuale: (rm04)

Vanno implementati:

```
@Override
public double getTotal() throws RemoteException {
    return 0;
}

@Override
public bool buyProduct(Product p, int qty) throws
RemoteException {
    return 0;
}
```

In effetti devo solo tornare un bool... errore...cambiamo..

..

Implementazione getTotal:

```
@Override
public double getTotal() throws RemoteException {
    double tot = 0;
    for (CartRow cr: rows){
        tot+=cr.product.cost * cr.qty;
    }
    return tot;
}
```

..

Implementazione buyProduct:

```
@Override
public boolean buyProduct(Product product, int qty) {
    System.out.println("buying " + product.descr);
    if (qty>0){
        cartRows.add(new CartRow(product, qty));
    }
    return qty>0;
}

..
```

Dal client:

```
...
System.out.println("Remote products " + products);

stub.buyProduct(products.get(0), 10);
stub.buyProduct(products.get(1), 20);

double tot = stub.getTotal();
System.out.println("Remote tot " + tot);
```

..

Appare:

```
Remote tot 50.0
```

Attenzione!

Se rilanciamo il client, i totali appaiono errati!

..

Attenzione!

Perche NON resettiamo il Cart!

Aggiugiamo all' interfaccia:

```
void resetCart() throws RemoteException;
....
```

Nel server:

```
@Override
public void resetCart() {
    cartRows = new ArrayList<>();
}
```

Nel client:

....

```
Sellable stub = (Sellable) registry.lookup("Sellable");
// Calling the remote method using
stub.resetCart();
```