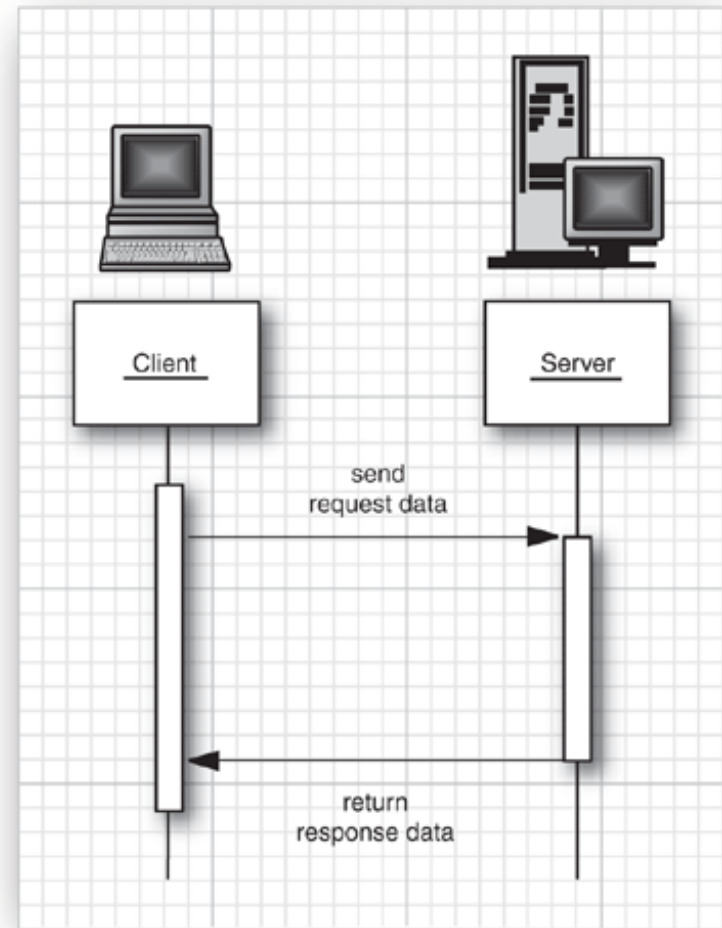


Programmazione
distribuita in Java:
Remote Method Invocation (RMI)

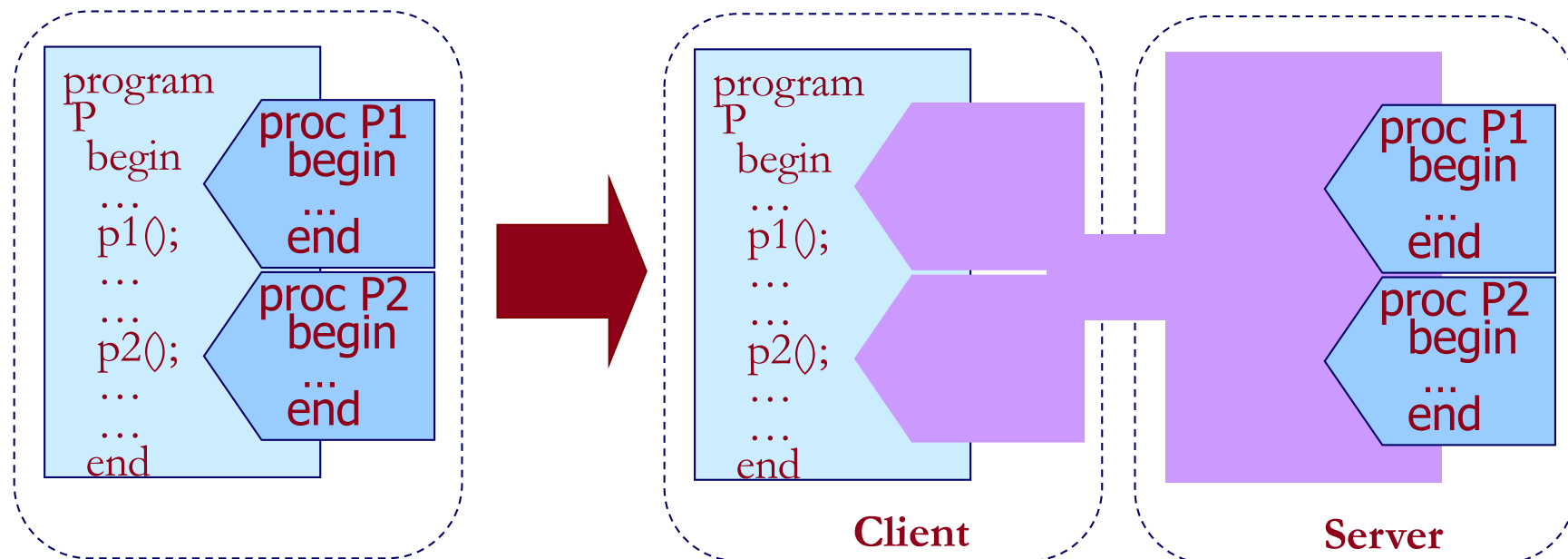
Verso RMI...

- 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?

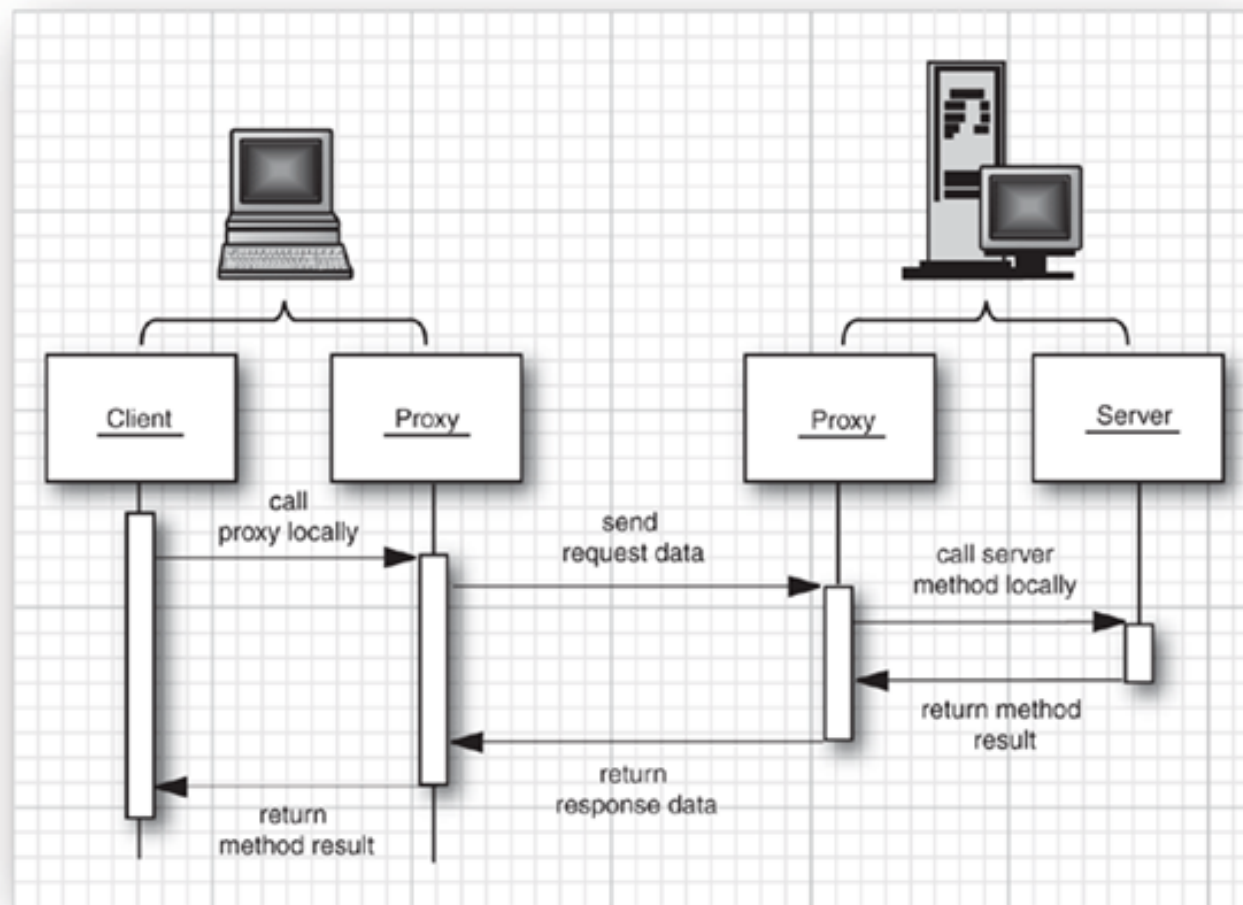
- L'illusione che la rete non esistesse, e che le invocazioni a metodo funzionassero anche su oggetti "remoti"



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
- Per farlo la soluzione tecnologica è quella di installare un **proxy sul client**
 - Il proxy appare al client come un normale oggetto
 - ...ma maschera tutto il processo di utilizzo della rete per eseguire il metodo sul server
- Allo stesso modo il programmatore che implementa il servizio non vuole preoccuparsi della gestione della comunicazione con il client
 -e per questo installa anche lui un **proxy sul server**
 - Il proxy del server comunica con il proxy del client creando un livello di astrazione al programmatore che non “vede” la rete

Verso RMI...



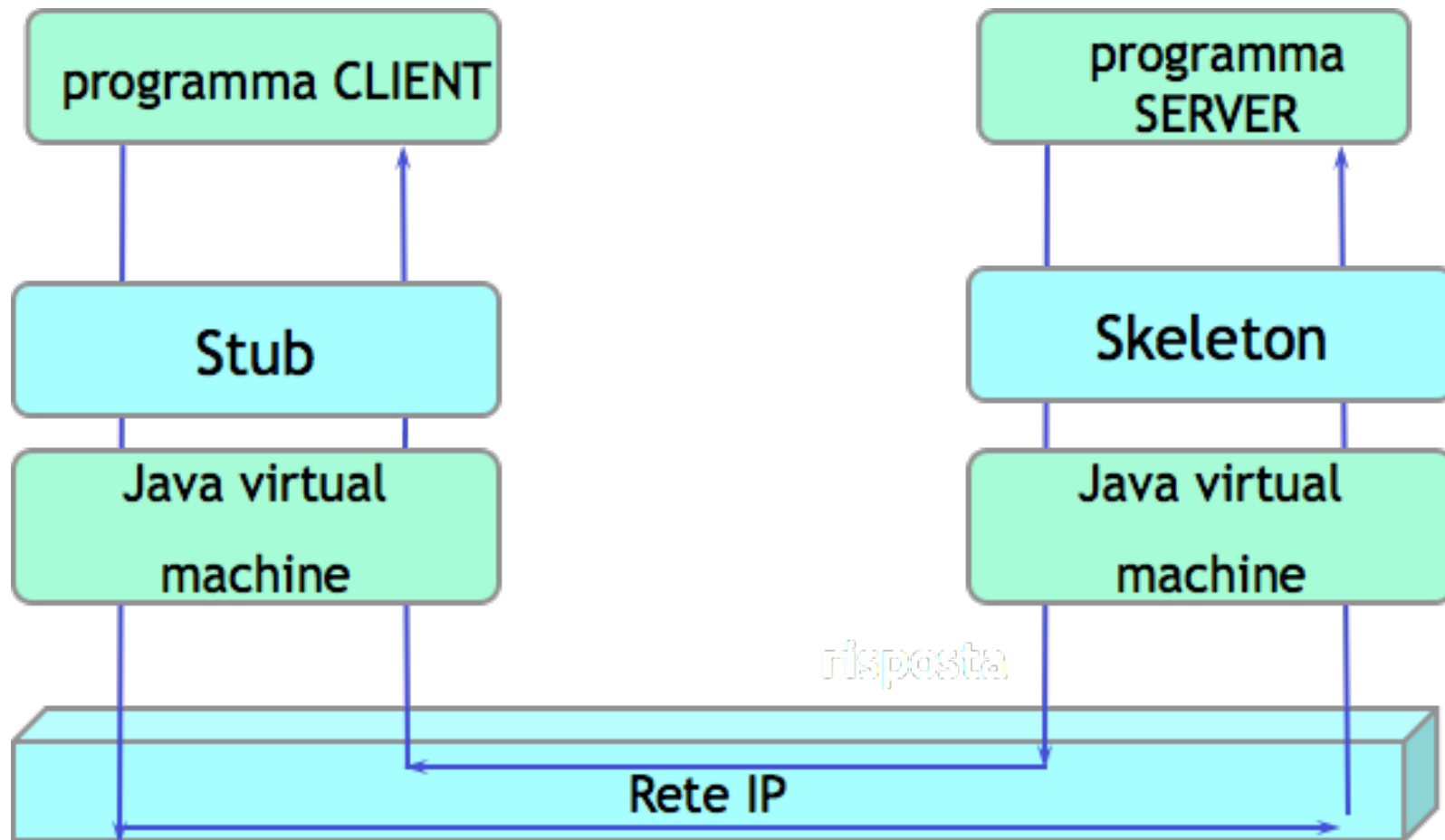
Le basi di RMI

- **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
 - E' capace di fare forward di chiamate di metodi attraverso la rete
- Esiste anche un proxy del client sul lato server, detto **skeleton**
 - E' una rappresentazione del client
 - Chiama i servizi del server
 - Sa come fare forward dei risultati attraverso la rete

Architettura interna

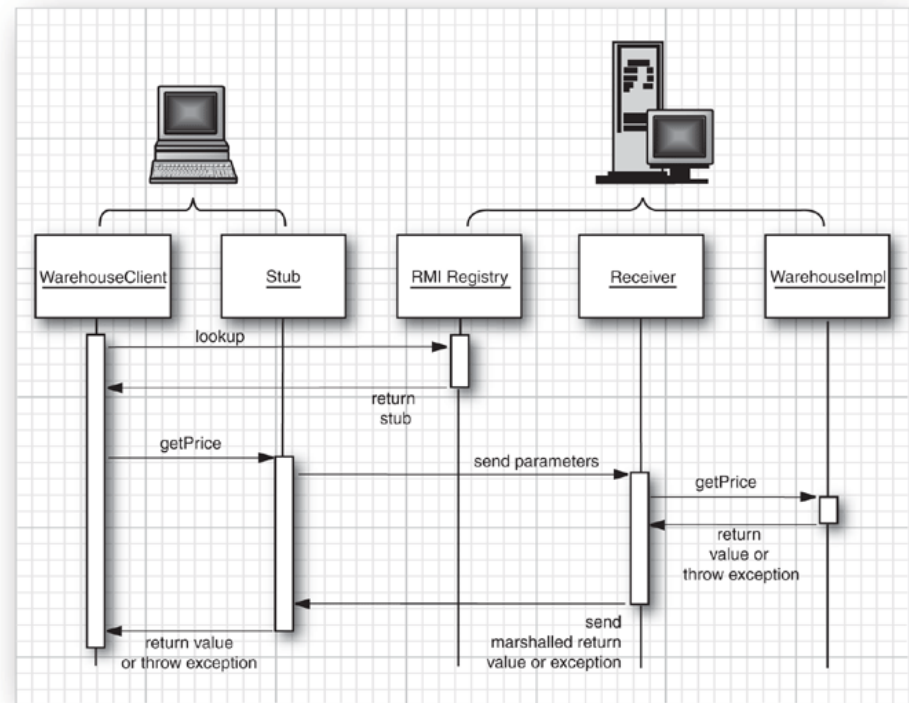


Documentazione ufficiale

- 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

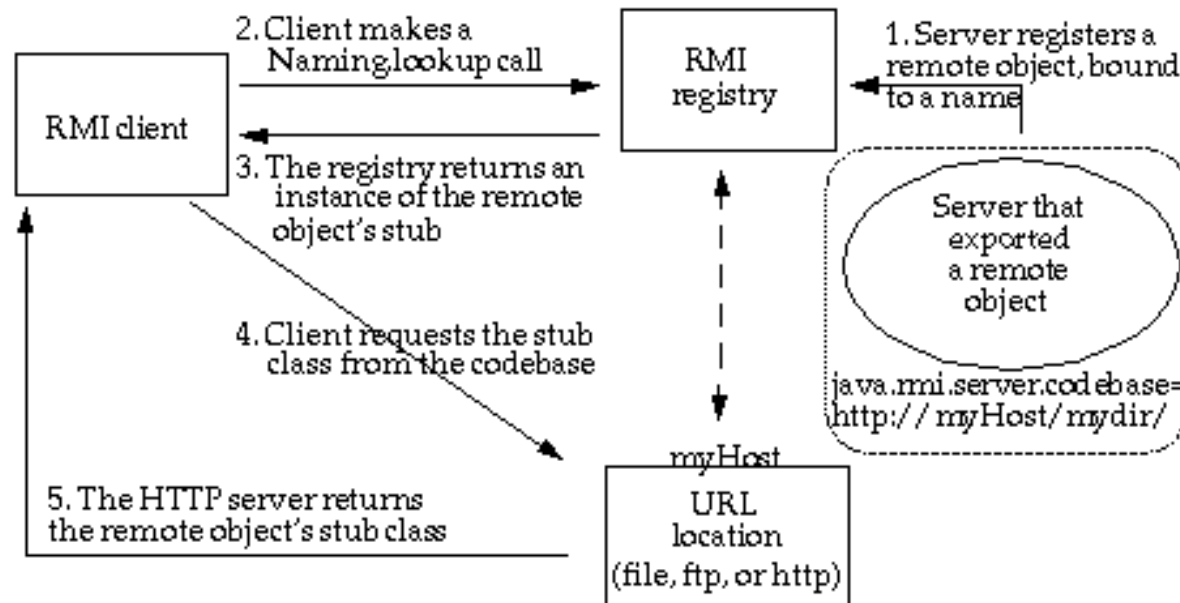
RMI Registry

- 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



Scaricare lo Stub

- Il registro RMI per spedire lo stub al client ha diverse opzioni
 - Se il client e il server risiedono sulla stessa macchina è possibile indicare al client il path locale per lo stub
 - Se il client e il server risiedono su macchine differenti è necessario utilizzare un server HTTP per permettere al registro di spedire lo stub



Running Example

- Vogliamo realizzare un applicazione che gestisce un magazzino (Warehouse)
- Il magazzino contiene un insieme di prodotti e ogni prodotto è identificato da:
 - Una stringa che identifica il prodotto
 - Un prezzo

Interfaccia condivisa client-server

- L'oggetto remoto Warehouse definisce l'interfaccia tra client e server
- Estende la **Remote** interface di Java
- Richiede la gestione di eventuali **RemoteException** nel caso in cui ci fossero errori di rete

Interfaccia condivisa client-server

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

```
public interface Warehouse extends Remote {  
    double getPrice(String description) throws RemoteException;  
}
```

Interfaccia condivisa dal client e dal server, entrambi sanno che si tratta di un'interfaccia remota

I client saranno costretti a gestire gli errori che possono sorgere durante l'invocazione di un oggetto remoto

Warehouse Server

- Il server implementa l'interfaccia remota
- Estende la classe **UnicastRemoteObject** che rende l'oggetto accessibile da remoto

Warehouse Server

```
import java.rmi.*;  
import java.rmi.registry.*;  
import java.rmi.server.*;  
import java.util.*;
```

Rende l'oggetto accessibile da remoto
(attraverso gli opportuni stub e skeletons)

```
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse {  
    private Map<String, Double> prices;
```

```
    public WarehouseImpl() throws RemoteException {  
        prices = new HashMap<String, Double>();  
        prices.put("Blackwell Toaster", 24.95);  
        prices.put("ZapXpress Microwave Oven", 49.95);  
    }
```

```
    public double getPrice(String description) throws RemoteException {  
        Double price = prices.get(description);  
        return price == null ? 0 : price;  
    }  
}
```

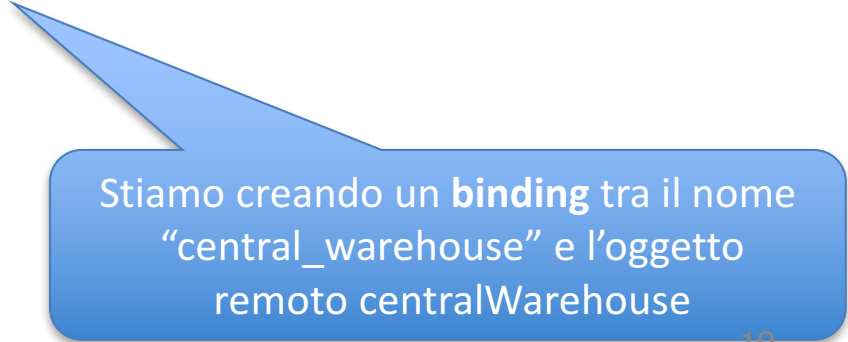
Definisco l'implementazione lato server
dell'interfaccia definita precedentemente

Pubblicare l'oggetto remoto

- All'avvio il server pubblica sul registro RMI l'oggetto remoto
- In questo modo il client potrà cercare gli oggetti remoti disponibili e ottenere un riferimento
- Il registro RMI deve essere online prima di avviare il server
 - Di default il registro si trova in localhost sulla porta 1099
 - Vedremo poi come lanciarlo...

```
WarehouseImpl centralWarehouse = new WarehouseImpl();  
Registry registry = LocateRegistry.getRegistry();
```

```
registry.bind("central_warehouse", centralWarehouse);
```



Stiamo creando un **binding** tra il nome "central_warehouse" e l'oggetto remoto centralWarehouse

Mettiamo tutto insieme...

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

public class WarehouseServer {
    public static void main(String[] args)
        throws RemoteException, AlreadyBoundException{
        System.out.println("Constructing server implementation...");
        WarehouseImpl centralWarehouse = new WarehouseImpl();


        System.out.println("Binding server implementation to registry...");
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("central_warehouse", centralWarehouse);

        System.out.println("Waiting for invocations from clients...");
    }
}
```

Note tecniche su bind()

- Per ragioni di sicurezza un'applicazione può associare, deassociare o riassociare un oggetto a un nome solo se l'applicazione gira sullo stesso host del registro
- Questo evita che client malevoli cambino le informazioni del registro
- I client possono fare un lookup degli oggetti...

```
String[] remoteObjects = registry.list();
```



Ritorna tutti i binding
attualmente presenti nel
registry...

Note tecniche sul registro

- Il registro è anch'esso un oggetto remoto
- Il metodo `bind()` appartiene all'interfaccia remota implementata dal registro, infatti...

```
void bind(String name, Remote obj)  
    throws RemoteException, AlreadyBoundException, AccessException;
```

- I parametri della chiamata dovranno essere serializzati/deserializzati
- Il registro scaricherà a runtime la definizione dell'interfaccia remota (nel nostro caso Warehouse) per serializzare l'oggetto che gli stiamo passando

Warehouse Client

- Lato client possiamo ottenere un riferimento al nostro stub con questo codice, purché:
 - Il registro sia online
 - L'oggetto remoto sia stato già pubblicato dal server

```
String remoteObjectName = "central_warehouse";
```

```
Warehouse centralWarehouse =  
    (Warehouse) registry.lookup(remoteObjectName);
```

- Notare il casting a Warehouse
 - Perché non usiamo WarehouseImpl?
 - Client e Server hanno in comune solo Warehouse, l'interfaccia remota!
 - Il client non sa nemmeno cosa sia WarehouseImpl

Warehouse Client

```
import java.rmi.*; import java.rmi.registry.*;
import java.util.*;
import javax.naming.*;
public class WarehouseClient {
    public static void main(String[] args)
        throws NamingException, RemoteException, NotBoundException {
        Registry registry= LocateRegistry.getRegistry();

        System.out.print("RMI registry bindings: ");
        String[] e = registry.list();

        for (int i=0; i<e.length; i++)
            System.out.println(e[i]);

        String remoteObjectName = "central_warehouse";
        Warehouse centralWarehouse = (Warehouse) registry.lookup(remoteObjectName);

        String descr = "Blackwell Toaster";
        double price = centralWarehouse.getPrice(descr);
        System.out.println(descr + ": " + price);
    }
}
```

Da questo momento in avanti,
tutte le chiamate all'oggetto
remoto non saranno più
distinguibili da chiamate ad un
oggetto locale!

“Deployare” l’applicazione RMI

- Cosa ci serve per far partire il tutto
 - Avviamo il server HTTP
 - Per permettere al registro RMI di recuperare la definizione delle nostre interfacce remote
 - Avviamo il registro RMI
 - Per permettere al client di trovare gli oggetti remoti pubblicati dal nostro server
 - Avviamo il server
 - All’avvio il server registrerà l’oggetto remoto Warehouse
 - Il registro RMI scarica la definizione dell’interfaccia remota dal server HTTP
 - Avviamo il client
 - Per vedere finalmente l’output della nostra applicazione

Settings (1)

- Sulla macchina server avremo

server/

WarehouseServer.class

WarehouseImpl.class

download/

Warehouse.class

- Sulla macchina client avremo

client/

WarehouseClient.class

Settings (2)

- In fase di bind il registro RMI ha quindi bisogno di accedere alla definizione delle interfacce remote
- I file .class solitamente vengono distribuiti con un normale web server
 - Nel nostro esempio il server deve rendere disponibile il file Warehouse.class
- Scarichiamo quindi NanoHTTPD web server
 - Un mini-server web la cui implementazione è contenuta tutta in NanoHTTPD.java
- Dopo aver compilato il server dovremmo trovarci nella seguente situazione

```
download/  
    Warehouse.class  
    NanoHTTPD.class
```

Avviamo l'HTTP Server

- Avviamo il server HTTP in localhost sulla porta 8080
- Per verificare che tutto funzioni proviamo ad accedere all'indirizzo `http://localhost:8080` via browser
- Tale indirizzo verrà poi utilizzato dal server per dichiarare la location della codebase RMI
 - Vedere poi l'avvio del server della nostra applicazione

```
$ java download/NanoHTTPD 8080
```

Avviamo il registro RMI

- Su Linux e OSX

`$ rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false`

- Su Windows

`$ start rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false`

- In generale è necessario che l'eseguibile `rmiregistry` sia presente nel path
 - Viene distribuito con Java, lo trovate quindi nella sua cartella di installazione

Avviamo il Server

- Andiamo nella directory del server e digitiamo

```
$ java -Djava.rmi.server.useCodebaseOnly=false  
      -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
```

- L'opzione `-Djava.rmi.server.codebase` serve a specificare l'URL dove si trovano i file `.class` che servono al registro RMI
- Quando eseguite questo comando date un'occhiata al terminale dove sta girando NanoHTTPD
 - Vedrete un messaggio che mostra informazioni sulla richiesta del registro RMI interessato al file `WareHouse.class`

Nota tecnica

- Dopo aver avviato il server la console resta in esecuzione
- Se guardiamo il codice del server questo potrebbe sembrarci strano
 - Il programma ha solo creato un oggetto WarehouseImpl e l'ha registrato sul RMI registry
 - La spiegazione è che quando creiamo un oggetto di tipo UnicastRemoteObject viene creato un thread separato che tiene vivo il programma
 - Questa funzione è necessaria per assolvere alla funzione di server

Avviamo il client

- Infine apriamo una quarta console, andiamo nella cartella contenente i file del client e digitiamo

```
$ java WarehouseClient
```

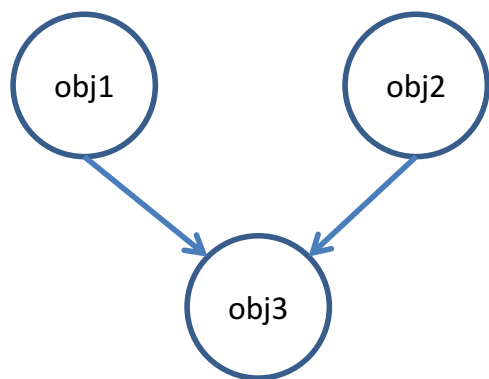
- Questo completa la nostra prima applicazione RMI!

Passaggio di oggetti

- Un oggetto **non-remoto**, passato come parametro, o restituito come risultato da un metodo remoto, è sempre passato per copia
 - Ovvero serializzato, scritto nello stream, e ricaricato all'altro estremo dello stream, ma come un oggetto differente
 - Modificare quindi un oggetto ricevuto mediante invocazione remota **non ha alcun effetto** sull'istanza originale di chi l'ha inviato
- Un oggetto **remoto**, già esportato, passato come parametro, o restituito come risultato da un metodo remoto è passato mediante il suo **stub**
 - Un oggetto remoto passato come parametro può solo implementare interfacce remote

Referential Integrity

- Se due riferimenti ad un oggetto sono passati da una JVM ad un'altra utilizzando una **singola** chiamata remota, questi riferimenti punteranno allo stesso oggetto anche nella JVM ricevente
- All'interno di una stessa chiamata remota il sistema RMI mantiene la **referential integrity** tra gli oggetti passati come parametro o come valori di ritorno



Caso (1)

```
remoteObject.remoteMethodTwoParameters(obj1, obj2);
```

Caso (2)

```
remoteObject.remoteMethodOneParameter(obj1);  
remoteObject.remoteMethodOneParameter(obj2);
```


Concorrenza

- La specifica RMI prevede che il server possa eseguire le invocazioni dei metodi remoti in modalità **multi-threaded**
 - I metodi esposti a chiamate remote devono essere thread-safe
 - Gestire la concorrenza è a **carico del programmatore**

Dynamic Class Loading

- Mediante il dynamic class loading in Java è possibile caricare a runtime la definizione di classi Java
- Questa caratteristica è usata con RMI
 - Il client può ricevere da parte del server delle classi sconosciute per le quali è necessario scaricare la definizione corrispondente, cioè il file .class
- Vediamo con un esempio pratico un caso d'uso di questa tecnologia

Warehouse V2

- Vogliamo modificare il nostro progetto Warehouse affinché cerchi un determinato prodotto sul server sulla base di una lista di keyword e non più semplicemente grazie alla descrizione del prodotto
- Ecco l'interfaccia remota aggiornata:

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

```
public interface Warehouse extends Remote  
{  
    double getPrice(String description) throws RemoteException;  
    Product getProduct(List<String> keywords) throws RemoteException;  
}
```

La classe Product

```
import java.io.*;
```

```
public class Product implements Serializable {  
    private String description;  
    private double price;  
    private Warehouse location;  
  
    public Product(String description, double price) {  
        this.description = description;  
        this.price = price;  
    }  
    public String getDescription() {  
        return description;  
    }  
    public double getPrice(){  
        return price;  
    }  
    public Warehouse getLocation() {  
        return location;  
    }  
    public void setLocation(Warehouse location) {  
        this.location = location;  
    }  
}
```

- Questa classe sarà presente sia sul client che sul server
- Stabilisce cosa è un prodotto e che funzionalità offre
- Non è un oggetto remoto ma è **serializzabile**
- Il server dovrà inviare i prodotti al client

La classe Book

```
public class Book extends Product {  
  
    private String isbn;  
  
    public Book(String title, String isbn, double price) {  
        super(title, price);  
        this.isbn = isbn;  
    }  
  
    public String getDescription() {  
        return super.getDescription() + " " + isbn;  
    }  
  
}
```

Struttura progetto

- Il progetto è diviso in tre compilation units
- Server e Client dipendono entrambi dalla definizione delle interfacce condivise
 - Warehouse è l'interfaccia dell'oggetto remoto
 - Product è richiesto da Warehouse

server/

WarehouseServer.class
WarehouseImpl.class
Book.class

download/

Warehouse.class
Product.class

client/

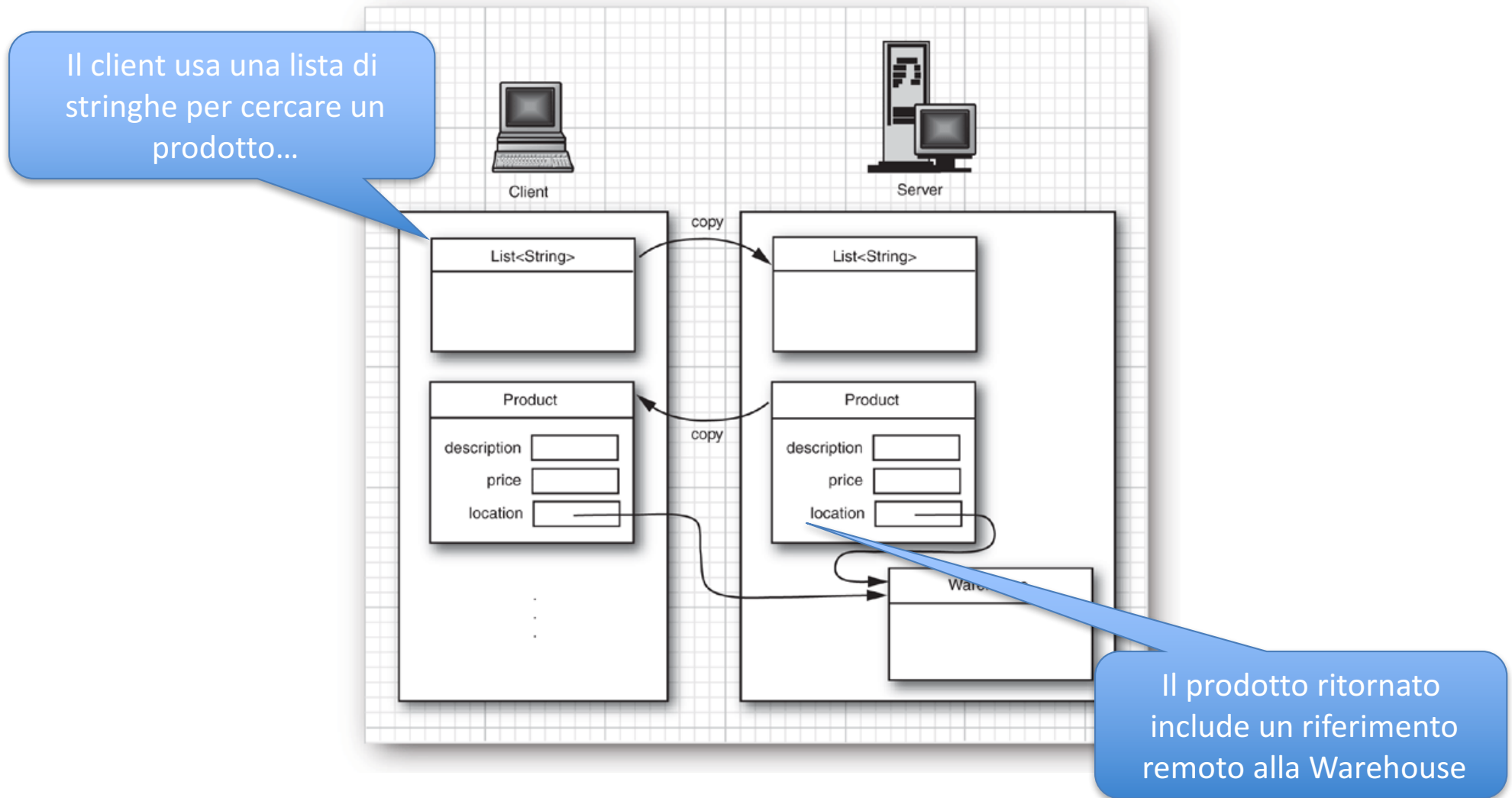
WarehouseClient.class

<<dependency>>

<<dependency>>

Book è un particolare tipo
di Product, inizialmente
non noto al Client...

Workflow dell'applicazione



WarehouseImpl

```
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse {

    private Map<String, Product> products;
    private Product backup;

    public WarehouseImpl(Product backup) throws RemoteException {
        products = new HashMap<String, Product>();
        this.backup = backup;
    }

    public void add(String keyword, Product product){
        product.setLocation(this);
        products.put(keyword, product);
    }

    public double getPrice(String description) throws RemoteException {
        for (Product p : products.values())
            if (p.getDescription().equals(description)) return p.getPrice();
        if (backup == null) return 0;
        else return backup.getPrice(description);
    }

    public Product getProduct(List<String> keywords) throws RemoteException {
        for (String keyword : keywords){
            Product p = products.get(keyword);
            if (p != null) return p;
        }
        return backup;
    }
}
```


WarehouseServer

```
import java.rmi.*;
import javax.naming.*;

public class WarehouseServer {
    public static void main(String[] args) throws RemoteException, NamingException {

        System.out.println("Constructing server implementation...");
        WarehouseImpl centralWarehouse = new WarehouseImpl(
            new Book("BackupBook", "123456", 66.99));

        centralWarehouse.add("toaster", new Product("Blackwell Toaster", 23.95));

        System.out.println("Binding server implementation to registry...");
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("central_warehouse", centralWarehouse);

        System.out.println("Waiting for invocations from clients...");
    }
}
```

WarehouseClient

```
import java.rmi.*;
import java.util.*;
import javax.naming.*;
import java.util.ArrayList;
public class WarehouseClient {
    public static void main(String[] args) throws NamingException, RemoteException {

        Context namingContext = new InitialContext();
        System.out.print("RMI registry bindings: ");
        Enumeration<NameClassPair> e =
            namingContext.list("rmi://localhost/");

        while (e.hasMoreElements())
            System.out.println(e.nextElement().getName());

        String url = "rmi://localhost/central_warehouse";
        Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);

        ArrayList<String> l=new ArrayList<String>();
        l.add("toaster");
        Product p=centralWarehouse.getProduct(l);
        System.out.println("Description: " + p.getDescription());
    }
}
```

Metodo alternativo per
ispezionare RMI
registry ed ottenere un
riferimento ad un
oggetto remoto

A runtime potremmo
ricevere un book! Ma
come fa il client a
conoscerlo? Book è
stato compilato
solamente sul server

Dynamic Class Loading

- Il nostro server torna dei prodotti sulla base delle keyword che ha inviato il client
- Tuttavia se il prodotto non è presente si è deciso di tornare un oggetto di backup
 - ...in questo caso, un oggetto di tipo Book che estende Product
- Ma il client non ha idea di cosa sia un Book
 - Book non è stato inserito tra le dipendenze del Client
 - Quando abbiamo compilato il client Book non era richiesto
 - Il codice del Client non ha riferimenti a Book, quindi compila anche senza

Dynamic Class Loading

- Il server comunica l'URL della codebase al client
 - Mediante l'attributo `java.rmi.server.codebase`
- Il client contatta quindi il server HTTP e scarica il file `Book.class` in modo da poter eseguire il suo codice
 - Tutto questo avviene in maniera **trasparente e automatica!**