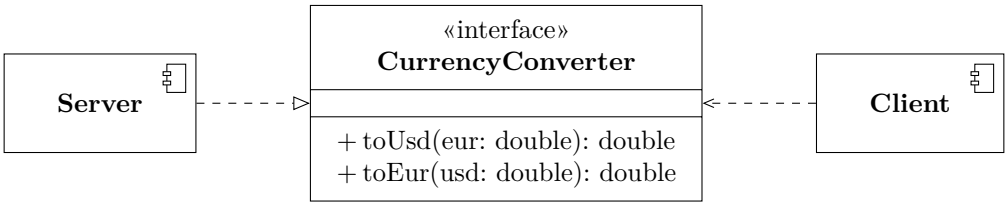
Azzolini Riccardo

Esempi di programmazione con RMI

1 Convertitore di valuta

Quest'applicazione realizza un semplice convertitore di valuta, che permette di convertire un valore da euro a dollari e viceversa. La conversione viene eseguita dal server, che poi restituisce il risultato al client.

L'organizzazione dei componenti dell'applicazione è quella mostrata nella seguente figura:



1.1 Interfaccia remota

Come al solito, è necessario definire un'interfaccia remota che descriva il servizio offerto dal server:

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

public interface CurrencyConverter extends Remote {
    double toEur(double usd) throws RemoteException;
    double toUsd(double eur) throws RemoteException;
}
```

1.2 Implementazione del server

Per implementare il server, questa volta, si sceglie (a scopo illustrativo) di *non* estendere `UnicastRemoteObject`:

```java
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CurrencyConverterImpl implements CurrencyConverter {
    private static final double USD_TO_EUR = 0.895415473;
    private static final double EUR_TO_USD = 1.114365;

    public double toEur(double usd) {
        return usd * USD_TO_EUR;
    }

    public double toUsd(double eur) {
        return eur * EUR_TO_USD;
    }

    public static void main(String[] args) throws RemoteException {
        CurrencyConverterImpl obj = new CurrencyConverterImpl();
        CurrencyConverter stub = (CurrencyConverter)
            UnicastRemoteObject.exportObject(obj, 3939);
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("CurrencyConverter", stub);
        System.out.println("Server ready");
    }
}
```

Invece, l'oggetto viene reso referenziabile da remoto chiamando, nel main, il metodo statico `exportObject`, che ha due argomenti:

- l'oggetto da rendere remoto;
- il numero della porta sulla quale l'oggetto potrà ricevere chiamate remote.[1]

1.3 Implementazione del client

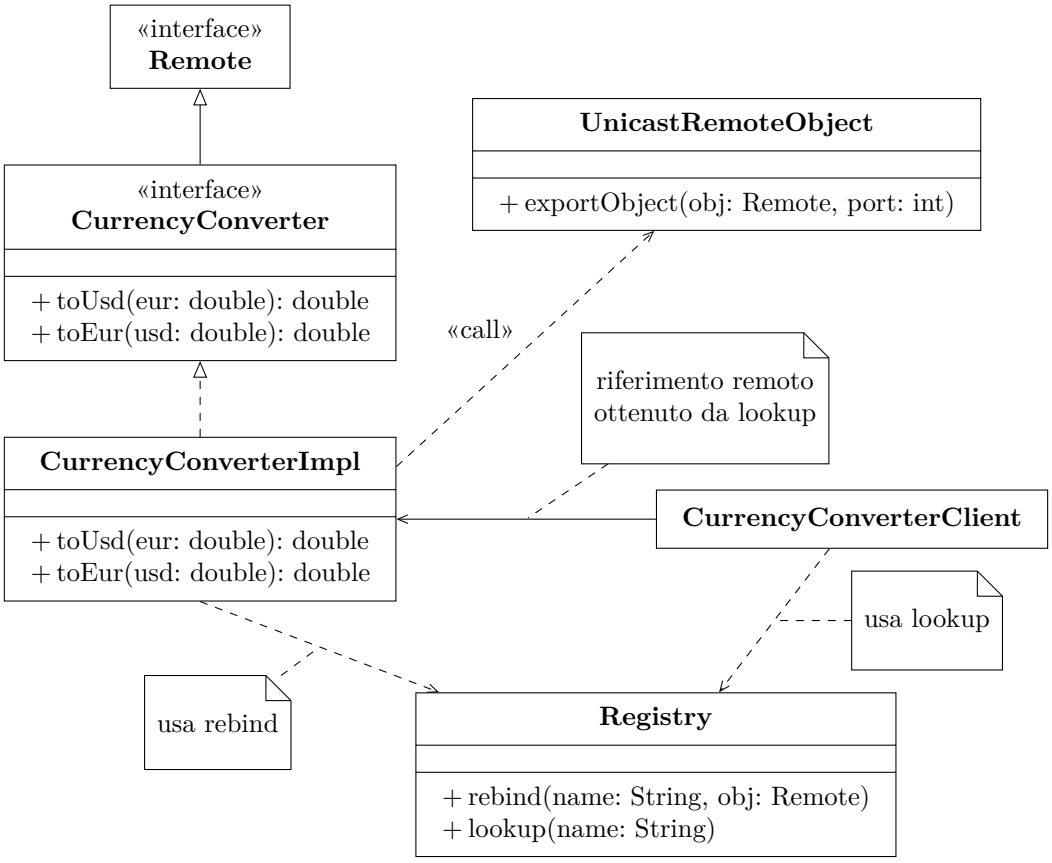
Il client riceve opzionalmente come argomento l'indirizzo del server (mentre, se non ci sono argomenti, di default usa localhost), ma, per il resto, viene implementato come al solito:

[1] Se si vuole che la porta venga scelta automaticamente, come negli esempi visti in precedenza, nei quali si estendeva `UnicastRemoteObject`, si può passare come argomento il valore 0. Viceversa, è possibile specificare la porta anche quando si estende tale classe, invocando un suo apposito costruttore mediante una chiamata `super(port)`.

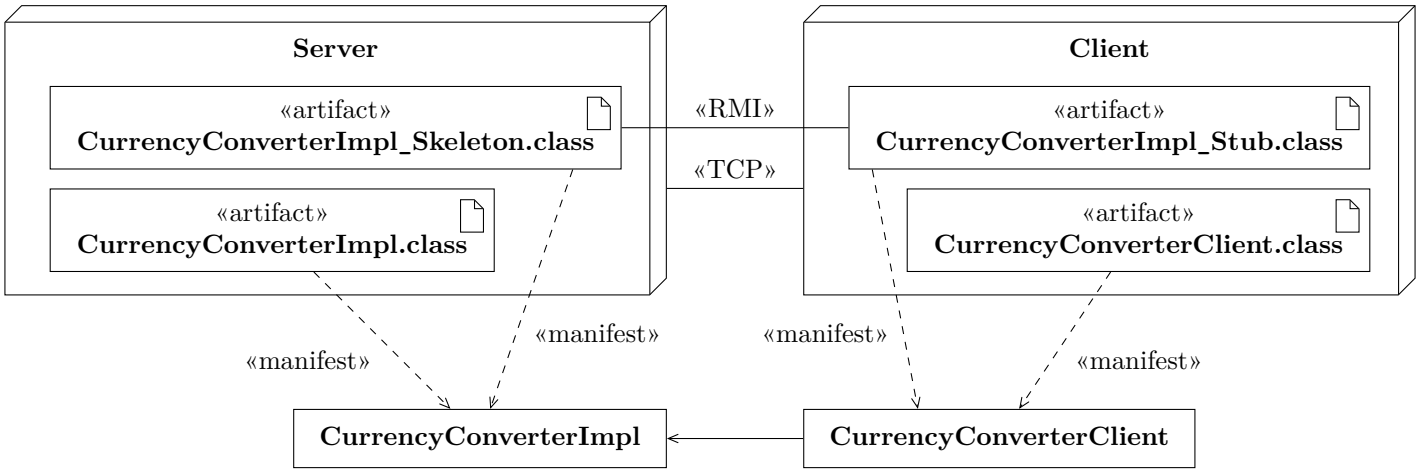
```java
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

public class CurrencyConverterClient {
    public static void main(String[] args) throws Exception {
        String host = args.length >= 1 ? args[0] : null;
        Registry registry = LocateRegistry.getRegistry(host);
        CurrencyConverter stub = (CurrencyConverter)
            registry.lookup("CurrencyConverter");

        for (int usd = 1; usd < 10; usd++) {
            System.out.println(
                usd + " USD = "  + stub.toEur(usd) + " EUR"
            );
        }
        for (int eur = 1; eur < 10; eur++) {
            System.out.println(
                eur + " EUR = "  + stub.toUsd(eur) + " USD"
            );
        }
    }
}
```

1.4 Class e deployment diagram

Le relazioni tra le varie classi (e interfacce) di server e client sono mostrate in questo diagramma:



Invece, il seguente deployment diagram mostra dove sono fisicamente collocati e come comunicano i vari elementi dell'applicazione:

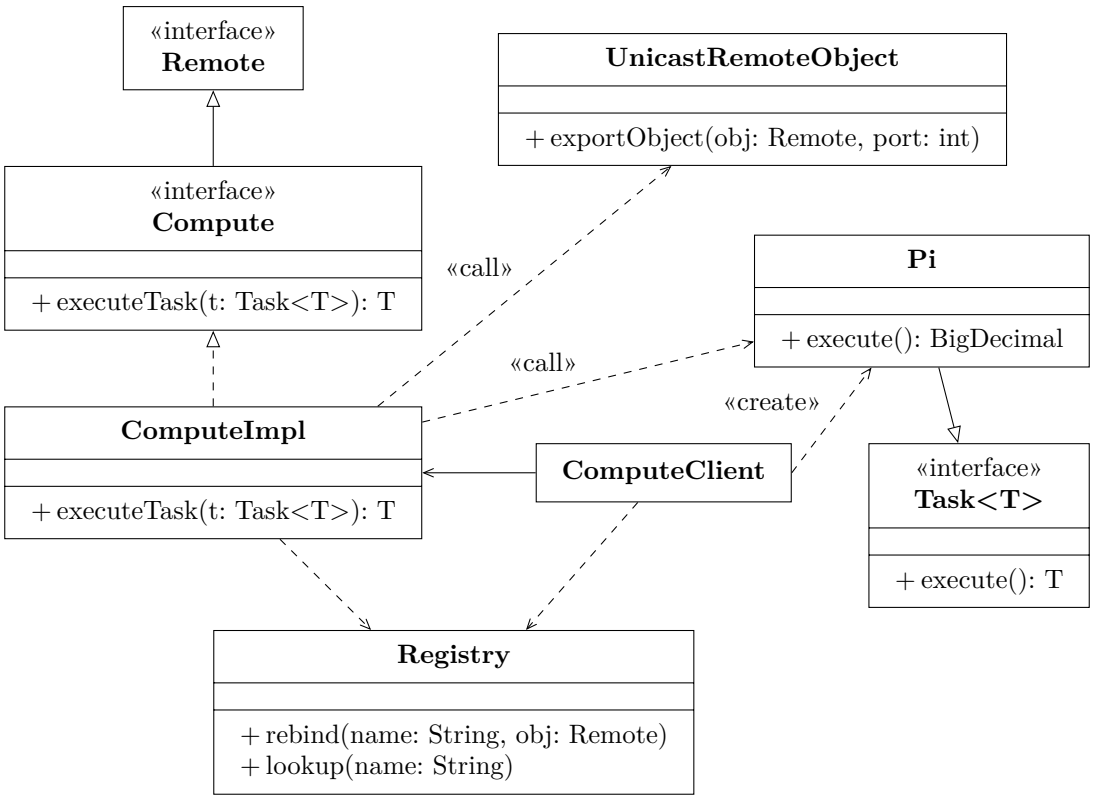


- Qui gli stub e gli skeleton sono mostrati esplicitamente, anche se, nelle versioni recenti di Java, la loro presenza è del tutto implicita (ma il funzionamento sottostante è lo stesso).
- Al livello applicativo, la comunicazione avviene tramite RMI tra stub e skeleton. Al livello di trasporto, RMI usa una connessione TCP (in particolare, in questo esempio, il server utilizza la porta 3939).
- Per semplicità, è stato omesso il registry.

2 Motore di calcolo

In questo secondo esempio, il server è un "motore di calcolo" (compute engine) che esegue elaborazioni pesanti per conto del client.

La vista a livello di classi del sistema è:



Essendo appunto una vista a livello di classi, essa non mostra, in particolare, che l'istanza `Pi` viene trasmessa dal client al server mediante la serializzazione, quindi la chiamata del metodo `execute` eseguita da `ComputeImpl` è una chiamata locale.

2.1 Interfacce `Task` e `Compute`

Per prima cosa, si definisce un'interfaccia generica `Task<T>`, la quale rappresenta un calcolo che restituisce un risultato di tipo `T`:

```
public interface Task<T> {
    T execute();
}
```

L'interfaccia remota ha un singolo metodo `executeTask`, che, come suggerisce il nome, richiede al server l'esecuzione di un task e, al termine, ne restituisce il risultato:

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

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

2.2 Implementazione del server

Il server implementa il metodo `Compute.executeTask` semplicemente eseguendo il task passato come argomento e restituendone il risultato:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ComputeImpl implements Compute {
    public <T> T executeTask(Task<T> t) {
        return t.execute(); // Chiamata locale
    }

    public static void main(String[] args) throws RemoteException {
        ComputeImpl engine = new ComputeImpl();
        Compute stub = (Compute)
            UnicastRemoteObject.exportObject(engine, 3939);
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("Compute", stub);
        System.out.println("Server ready");
    }
}
```

Quindi, in quest'applicazione, il server fornisce solo la potenza di calcolo, mentre il codice da eseguire è fornito dai client.

2.3 Task da eseguire

Il task che il client chiederà al server di eseguire è rappresentato dalla classe `Pi`. Esso calcola il valore π (sotto forma di `BigDecimal`) fino al numero di cifre decimali specificato nel costruttore. Siccome il task deve essere trasmesso al server, questa classe è serializzabile.

```
import java.io.Serializable;
import java.math.BigDecimal;
import java.math.RoundingMode;

public class Pi implements Task<BigDecimal>, Serializable {
    private static final long serialVersionUID = 1;
    private final int digits;

    public Pi(int digits) {
        this.digits = digits;
    }

    public BigDecimal execute() {
        return computePi(digits);
    }

    private static BigDecimal computePi(int digits) {
        int scale = digits + 5;
        final BigDecimal FOUR = BigDecimal.valueOf(4);
        BigDecimal pi =
            arctanOfReciprocal(5, scale)
            .multiply(FOUR)
            .subtract(arctanOfReciprocal(239, scale))
            .multiply(FOUR);
        return pi.setScale(digits, RoundingMode.HALF_UP);
    }

    private static BigDecimal arctanOfReciprocal(
        int reciprocal, int scale
    ) {
        final RoundingMode ROUNDING_MODE = RoundingMode.HALF_EVEN;
        BigDecimal x = BigDecimal.valueOf(reciprocal);
        BigDecimal x2 = x.pow(2);
```

```
        BigDecimal numerator =
            BigDecimal.ONE.divide(x, scale, ROUNDING_MODE);
        BigDecimal result = numerator;

        BigDecimal term;
        int i = 1;
        do {
            numerator = numerator.divide(x2, scale, ROUNDING_MODE);
            BigDecimal denominator = BigDecimal.valueOf(2 * i + 1);
            term = numerator.divide(denominator, scale, ROUNDING_MODE);
            if (i % 2 == 0) {
                result = result.add(term);
            } else {
                result = result.subtract(term);
            }
            i++;
        } while (term.compareTo(BigDecimal.ZERO) != 0);

        return result;
    }
}
```

Nota: Per chi fosse interessato, questo codice calcola il valore di π usando la formula di Machin,

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

e uno sviluppo in serie di potenze dell'arcotangente,

$$\arctan(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{2i+1}$$

svolto fino alla precisione richiesta.

2.4 Implementazione del client

Il client crea un'istanza del task `Pi` (con un numero di cifre da calcolare che può essere passato come argomento, altrimenti di default è 100), la fa eseguire al server, e stampa il risultato ricevuto da quest'ultimo:

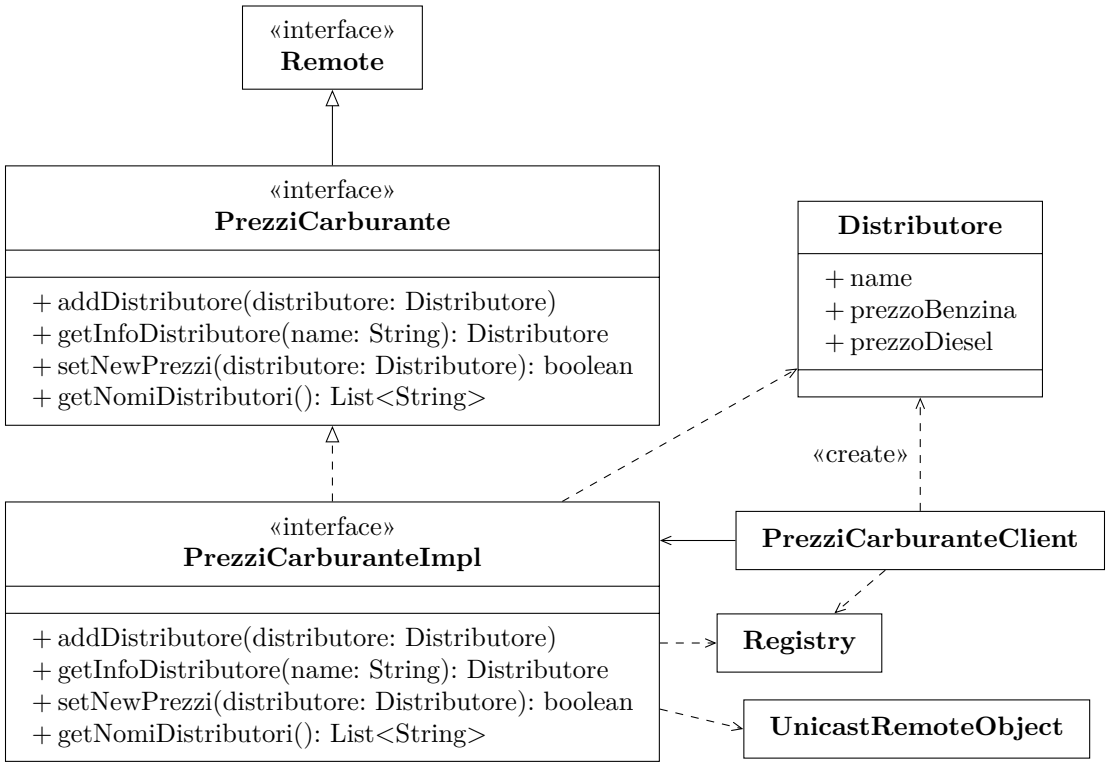
```
import java.math.BigDecimal;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
```

```
public class ComputeClient {
    public static void main(String[] args) throws Exception {
        String host = args.length >= 1 ? args[0] : null;
        int digits = args.length >= 2 ? Integer.parseInt(args[1]) : 100;

        Registry registry = LocateRegistry.getRegistry(host);
        Compute engine = (Compute) registry.lookup("Compute");
        Pi task = new Pi(digits);
        BigDecimal pi = engine.executeTask(task);
        System.out.println(pi);
    }
}
```

3 Prezzi del carburante

Quest'ultimo esempio implementa un database che memorizza i prezzi del carburante relativi a vari distributori: il server si occupa della memorizzazione, mentre il client inserisce, aggiorna e richiede i dati.

Il class diagram che rappresenta il sistema è:



3.1 Interfaccia remota

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

public interface PrezziCarburante extends Remote {
    void addDistributore(Distributore distributore)
        throws RemoteException;
    Distributore getInfoDistributore(String name)
        throws RemoteException;
    boolean setNewPrezzi(Distributore distributore)
        throws RemoteException;
    List<String> getNomiDistributori()
        throws RemoteException;
}
```

- `addDistributore` aggiunge un nuovo distributore, o aggiorna i prezzi se il distributore esiste già.
- `getInfoDistributore` restituisce le informazioni relative al distributore avente il nome passato come argomento.
- `setNewPrezzi` consente di aggiornare i prezzi del distributore dato; esso restituisce `true` se l'aggiornamento va a buon fine, o `false` se invece il distributore da aggiornare non esiste.
- `getNomiDistributori` restituisce la lista dei nomi dei distributori noti al server.

3.2 Classe `Distributore`

Le informazioni relative a un distributore (nome, prezzo della benzina e prezzo del gasolio) sono rappresentate da un'apposita classe serializzabile:

```
import java.io.Serializable;

public class Distributore implements Serializable {
    private static final long serialVersionUID = 1;
    private final String name;
    private double prezzoBenzina;
    private double prezzoDiesel;

    public Distributore(
        String name, double prezzoBenzina, double prezzoDiesel
    ) {
```

```
        this.name = name;
        this.prezzoBenzina = prezzoBenzina;
        this.prezzoDiesel = prezzoDiesel;
    }

    public String getName() {
        return name;
    }

    public double getPrezzoBenzina() {
        return prezzoBenzina;
    }

    public void setPrezzoBenzina(double prezzoBenzina) {
        this.prezzoBenzina = prezzoBenzina;
    }

    public double getPrezzoDiesel() {
        return prezzoDiesel;
    }

    public void setPrezzoDiesel(double prezzoDiesel) {
        this.prezzoDiesel = prezzoDiesel;
    }

    public String toString() {
        return "Distributore: " + name
            + "\n\tbenzina: " + prezzoBenzina
            + "\n\tdiesel:  " + prezzoDiesel;
    }
}
```

3.3 Implementazione del server

Il server memorizza i distributori in una `Map`, che associa a ciascun nome il distributore corrispondente:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
```

```
public class PrezziCarburanteImpl implements PrezziCarburante {
    Map<String, Distributore> distributori = new HashMap<>();

    public void addDistributore(Distributore distributore) {
        if (distributori.containsKey(distributore.getName())) {
            setNewPrezzi(distributore);
        } else {
            distributori.put(distributore.getName(), distributore);
            System.out.println(
                "aggiunto nuovo distributore " + distributore.getName()
            );
        }
    }

    public Distributore getInfoDistributore(String name) {
        return distributori.get(name);
    }

    public boolean setNewPrezzi(Distributore distributore) {
        Distributore esistente =
            distributori.get(distributore.getName());
        if (esistente == null) {
            System.err.println(
                "setNewPrezzi: il distributore "
                + distributore.getName() + " non esiste"
            );
            return false;
        }
        esistente.setPrezzoBenzina(distributore.getPrezzoBenzina());
        esistente.setPrezzoDiesel(distributore.getPrezzoDiesel());
        System.out.println(
            "aggiornati prezzi per distributore "
            + distributore.getName()
        );
        return true;
    }

    public List<String> getNomiDistributori() {
        return new ArrayList<>(distributori.keySet());
    }

    public static void main(String[] args) throws RemoteException {
        PrezziCarburanteImpl obj = new PrezziCarburanteImpl();
        PrezziCarburante stub = (PrezziCarburante)
```

```
            UnicastRemoteObject.exportObject(obj, 3939);
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("Prezzi", stub);
        System.out.println("Server ready");
    }
}
```

3.4 Implementazione del client

Il client simula un "viaggio", fermandosi ogni volta a un distributore casuale, che aggiunge al database se non è già presente, altrimenti ne aggiorna i prezzi. Al termine di ogni tappa, viene mostrato il numero di distributori salvati, poi si esegue uno sleep per simulare il tempo che trascorre prima di arrivare a un altro distributore.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.List;
import java.util.Random;

public class PrezziCarburanteClient {
    private static final Random random = new Random();

    public static void main(String[] args) throws Exception {
        String host = args.length >= 1 ? args[0] : null;
        Registry registry = LocateRegistry.getRegistry(host);
        PrezziCarburante stub = (PrezziCarburante)
            registry.lookup("Prezzi");

        int numTappe = random.nextInt(20) + 1;
        for (int tappa = 0; tappa < numTappe; tappa++) {
            String name = "d" + random.nextInt(10);
            Distributore distr = stub.getInfoDistributore(name);

            if (distr == null) {
                System.out.println(name + " sconosciuto: aggiungo!");
                stub.addDistributore(generaPrezzi(name));
            } else {
                System.out.println("prezzi vecchi: " + distr);
                stub.setNewPrezzi(generaPrezzi(name));
                Distributore newDistr = stub.getInfoDistributore(name);
                System.out.println("prezzi nuovi: " + newDistr);
            }
```

```
            List<String> names = stub.getNomiDistributori();
            System.out.println(
                "Numero distributori salvati: " + names.size()
            );
            Thread.sleep(random.nextInt(1000));
        }
    }

    private static Distributore generaPrezzi(String name) {
        return new Distributore(
            name,
            random.nextDouble() + 1,
            random.nextDouble() + 1
        );
    }
}
```

3.5 Come fermare ordinatamente il server

Se si arresta il server semplicemente uccidendo il processo, è possibile qualcosa resti in uno stato inconsistente. Allora, è preferibile aggiungere al server un comando che ne provochi lo spegnimento ordinato:[2]

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

public interface PrezziCarburante extends Remote {
    void addDistributore(Distributore distributore)
        throws RemoteException;
    Distributore getInfoDistributore(String name)
        throws RemoteException;
    boolean setNewPrezzi(Distributore distributore)
        throws RemoteException;
    List<String> getNomiDistributori()
        throws RemoteException;
    void shutdown() throws RemoteException;
}
```

Il nuovo comando viene invocato alla fine del main del client (in questo esempio, il server gestisce un singolo client, dopo di che può terminare):

[2] In altri casi, il server potrebbe determinare autonomamente che si deve arrestare, invece di farlo su richiesta del client: allora, non sarebbe necessario aggiungere un comando di spegnimento all'interfaccia remota.

```
public class PrezziCarburanteClient {
    // ...

    public static void main(String[] args) throws Exception {
        // ...
        for (int tappa = 0; tappa < numTappe; tappa++) {
            // ...
        }

        stub.shutdown();
    }

    // ...
}
```

Il server implementa il comando `shutdown` come segue:

```
import java.rmi.NotBoundException;
// ...

public class PrezziCarburanteImpl implements PrezziCarburante {
    // ...

    public void shutdown() throws RemoteException {
        Registry registry = LocateRegistry.getRegistry();
        try {
            registry.unbind("Prezzi");
        } catch (NotBoundException e) {}
        UnicastRemoteObject.unexportObject(this, false);

        new Thread() {
            public void run() {
                System.out.println("Shutting down...");
                try {
                    sleep(2000);
                } catch (InterruptedException e) {}
                System.exit(0);
            }
        }.start();
    }

    // ...
}
```

1. Si chiama `registry.unbind` per rimuovere il servizio dal registry.

2. Si usa `UnicastRemoteObject.unexportObject` per fare in modo che l'oggetto non sia più remoto, permettendo così la terminazione del programma. Gli argomenti della chiamata indicano che:

   a) l'oggetto da rendere non più remoto è quello corrente (`this`);

   b) non si vuole forzare la terminazione del server se sono ancora in corso interazioni con i client (`false`).

3. Se il server reagisse immediatamente a un comando `shutdown` si fermerebbe mentre il client è ancora connesso al server (per la stessa chiamata remota a `shutdown`), e quest'ultimo potrebbe quindi ricevere un'eccezione. Allora, si avvia un thread che (essendo non daemon) "tiene in vita" il server[3] per il tempo necessario affinché l'esecuzione metodo `shutdown` si completi e il client si disconnetta.

[3] In realtà, avendo passato l'argomento `false` a `unexportObject`, a causa della chiamata remota `shutdown` ancora in corso il server semplicemente non terminerebbe. Quindi, piuttosto che tenere in vita il server, il thread ha lo scopo di provocarne effettivamente la terminazione (usando `System.exit`) dopo che il client si è disconnesso.