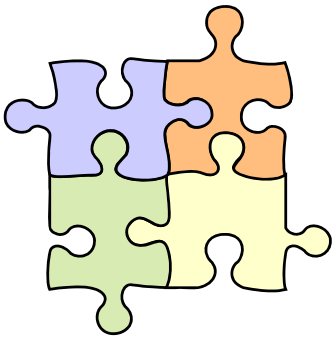
Luca Cabibbo



Architetture Software

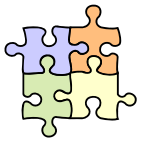
Oggetti distribuiti e invocazione remota

Dispensa ASW 830

ottobre 2014

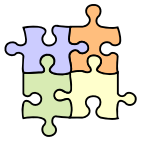
Il fatto che abbia funzionato in passato
non garantisce
che funzionerà adesso o in futuro.

Kenneth L. Cureton



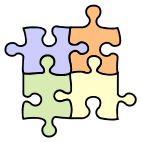
- Fonti

- [CDK/4e] Coulouris, Dollimore, Kindberg, Distributed Systems, Concepts and Design, 4th edition, 2005
  - Chapter 5, Distributed Objects and Remote Invocation
- [Liu] Liu, Distributed Computing – Principles and Applications, Pearson Education, 2004
  - Chapter 7, Distributed Objects
  - Chapter 8, Advanced RMI
- The Java Tutorial – Trail su RMI – *http://docs.oracle.com/javase/tutorial/rmi/index.html*

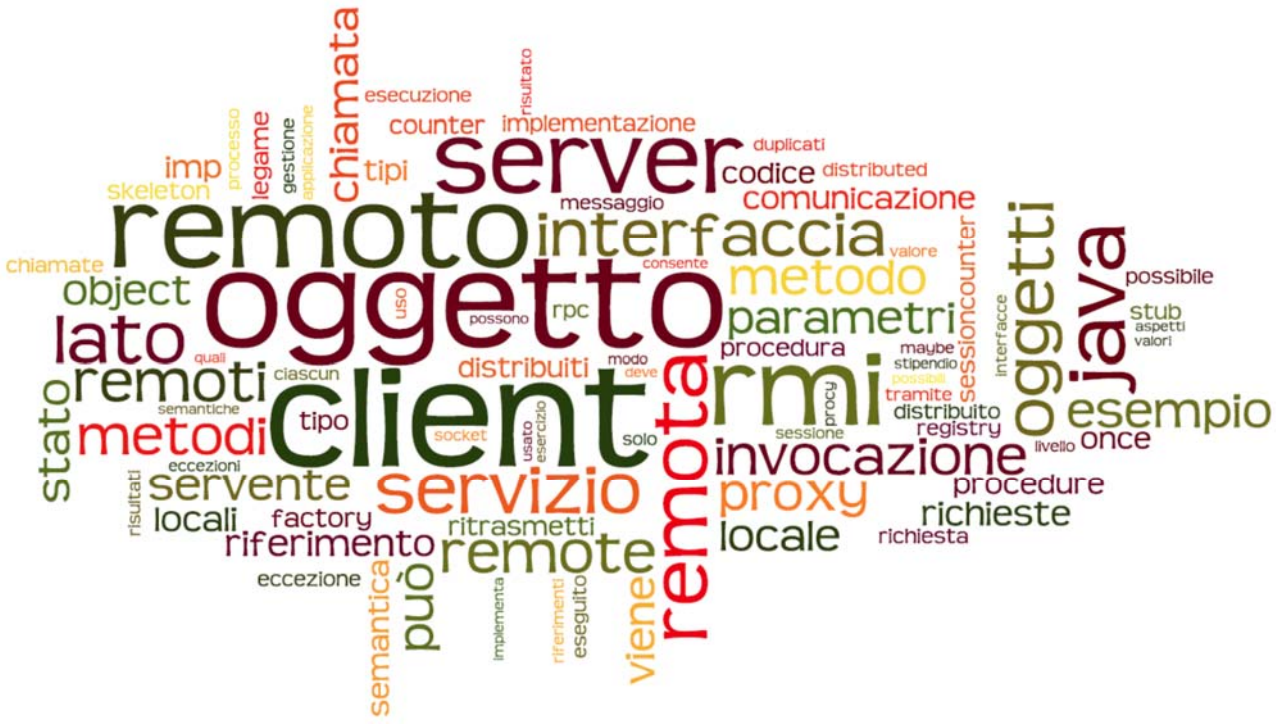


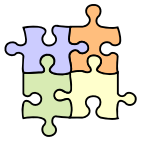
- Obiettivi e argomenti

- Obiettivi
  - comprendere alcuni aspetti fondamentali della programmazione a oggetti distribuiti
  - come studio di caso, introdurre Java RMI
- Argomenti
  - chiamata di procedure remote
  - oggetti distribuiti
  - Java RMI
  - architettura di Java RMI
  - confronto tra RMI e socket
  - Java RMI e connettori
  - ulteriori aspetti



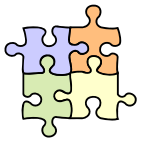
- Wordle





* Chiamata di procedure remote

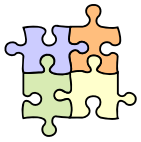
- I sistemi distribuiti possono essere programmati mediante dei meccanismi di programmazione di base – ad es., i socket
  - si tratta però di strumenti ad un basso livello di astrazione
    - il paradigma di comunicazione offerto è generico – è orientato allo scambio di dati/messaggi generici
  - sono strumenti difficili da usare, perché il programmatore deve controllare esplicitamente tutte le qualità richieste
    - ad es., gestione delle connessioni e dei fallimenti
- La programmazione dei sistemi distribuiti è oggi supportata da numerosi strumenti di middleware – uno tra i primi servizi di "comunicazione strutturata" è la *chiamata di procedure remote* – RPC
  - aumenta il livello di astrazione
  - comunicazione orientata alle "azioni", ovvero all'invocazione di sottoprogrammi



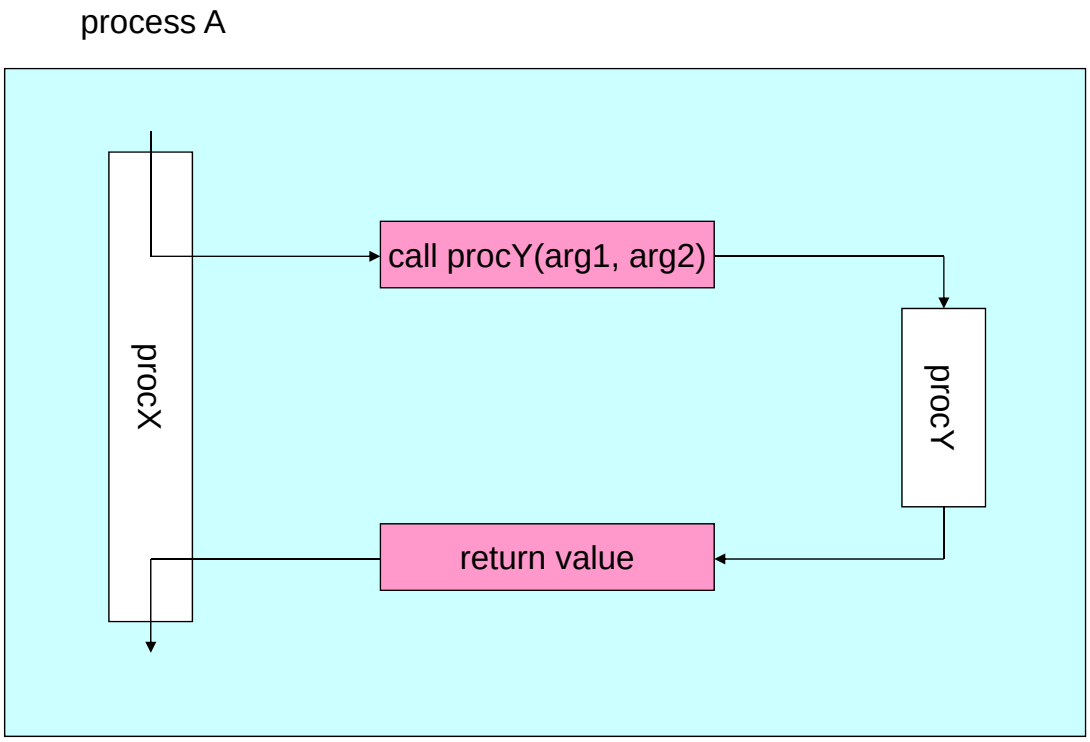
Remote Procedure Call

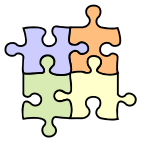
- *RPC* (*Remote Procedure Call*)
  - Remote Procedure Call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer) without the programmer explicitly coding the details for this remote interaction.
    That is, the programmer writes *essentially the same code* whether the subroutine is local to the executing program, or remote.
    [Wikipedia, ottobre 2013]

Attenzione, il fatto che sia scritto nello stesso modo non implica che funzioni nello stesso modo.

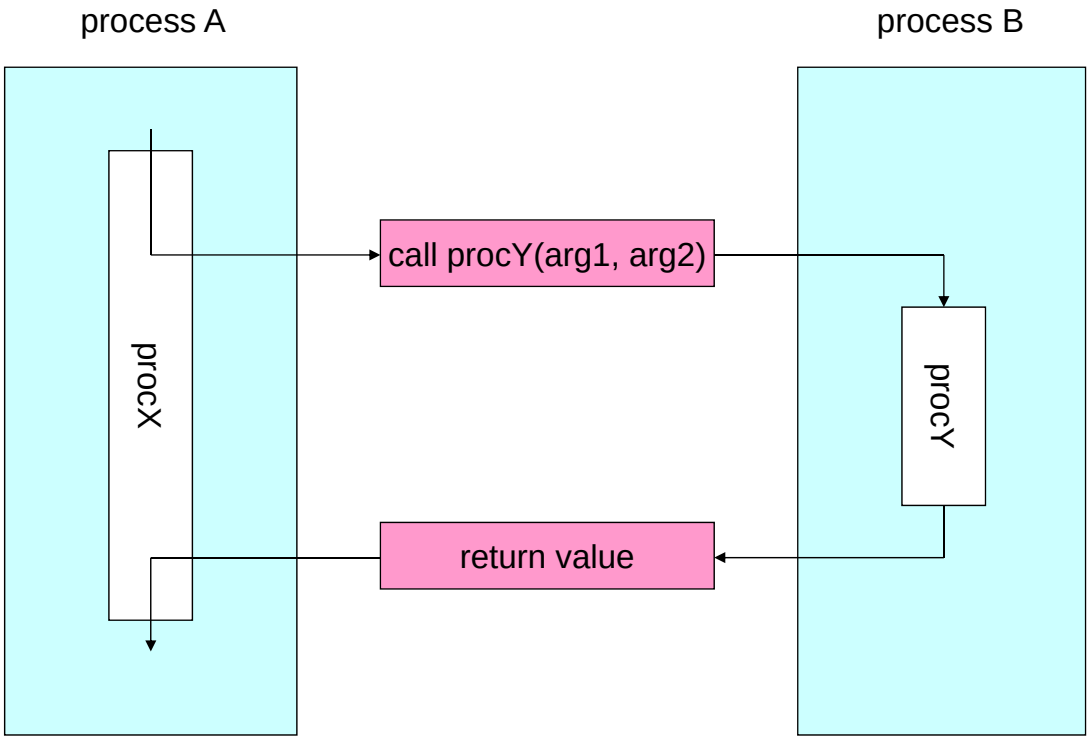


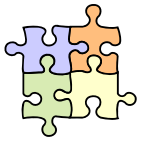
Una chiamata di procedura locale



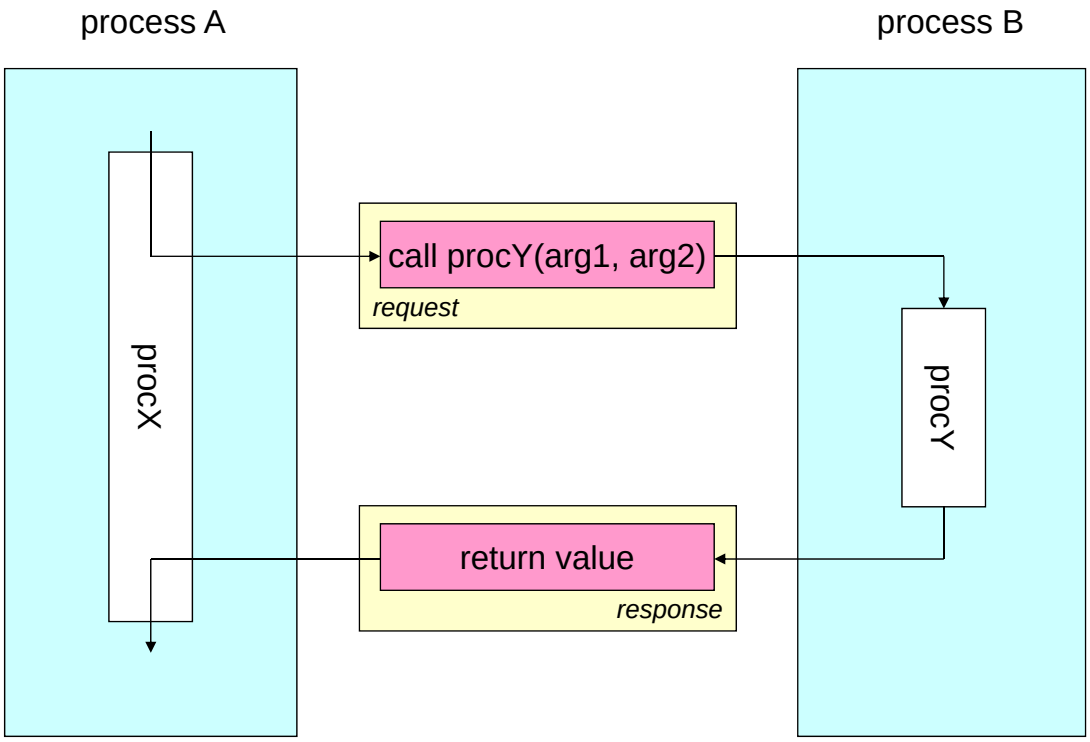


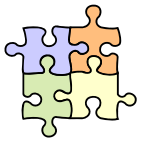
Una chiamata di procedura remota





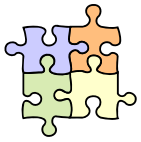
Una chiamata di procedura remota



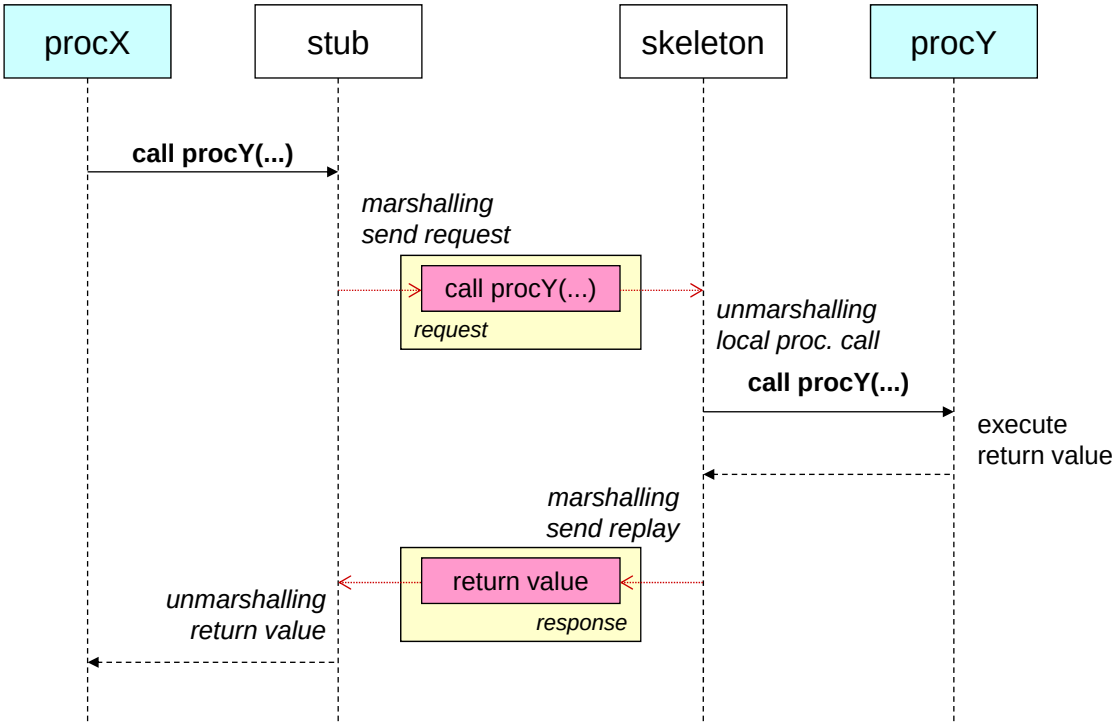


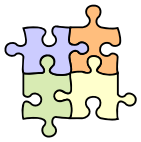
Paradigma dell'RPC

- La procedura remota ***procY*** viene chiamata (sintatticamente) da ***procX*** come se fosse una chiamata locale
  - in realtà, la procedura ***procY*** vive in un processo diverso da quello di ***procX***
  - in prima approssimazione, la modalità di esecuzione è basata su
    - legame dei parametri
    - esecuzione (remota) della procedura chiamata
    - restituzione del risultato



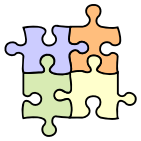
Realizzazione di RPC





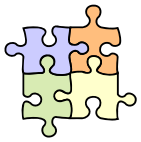
Realizzazione di RPC – stub

- La comunicazione tra le due procedure ***procX*** e ***procY*** avviene mediante un connettore realizzato come una coppia di moduli intermedi – che si occupano degli aspetti della distribuzione
  - il chiamante ***procX*** fa una chiamata locale a una procedura locale (anch'essa di nome ***procY***) a un modulo (*stub*) che è un rappresentante locale della vera procedura ***procY*** del server remoto
  - lo stub si occupa di effettuare la chiamata remota alla procedura remota ***procY***
    - effettua il marshalling dei parametri
    - invia un messaggio di richiesta al server tramite socket
  - inoltre lo stub
    - riceverà il messaggio di risposta alla chiamata remota
    - effettuerà l'unmarshalling dei risultati
    - restituirà questi risultati alla procedura chiamante ***procX***



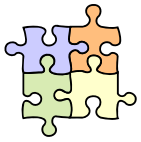
Realizzazione di RPC – skeleton

- In realtà, il messaggio inviato dallo stub non viene ricevuto direttamente dalla procedura remota ***procY***
  - piuttosto, il messaggio di richiesta viene ricevuto (tramite socket) da un modulo (*skeleton*) che è un rappresentante locale (lato server) della procedura chiamante
  - lo skeleton si occupa di effettuare la chiamata alla procedura (per lui locale) ***procY***
    - effettua l'unmarshalling della richiesta e dei parametri
    - effettua la chiamata locale alla procedura ***procY*** e riceve i risultati della chiamata
    - effettua il marshalling dei risultati
    - invia un messaggio di risposta al client tramite socket



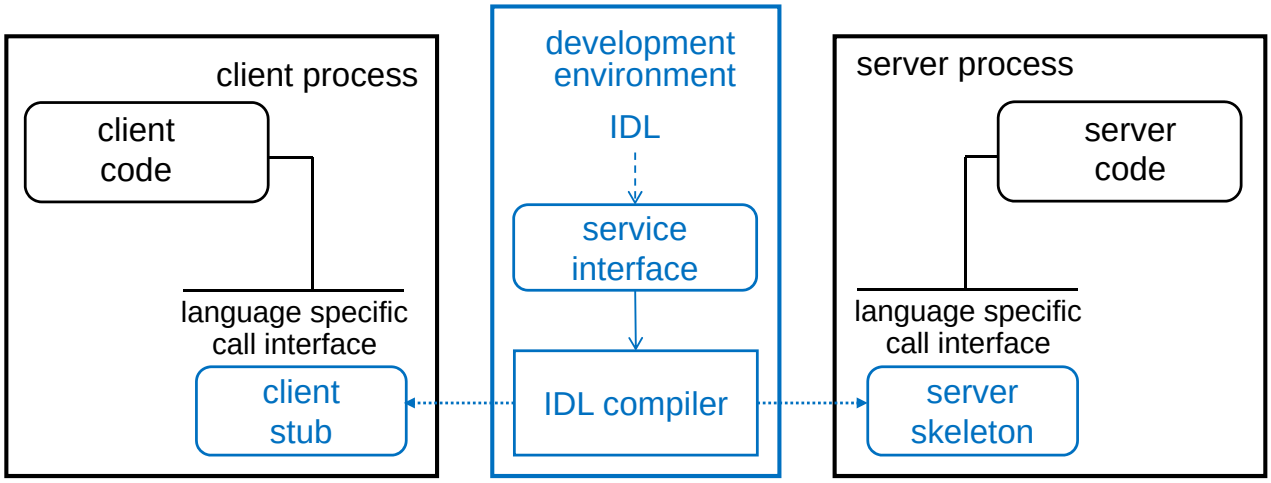
Discussione

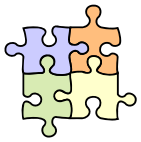
- Lo stub e lo skeleton
  - sono intermediari nella comunicazione – chiamati anche *proxy*
    - proxy lato client e proxy lato server
  - proxy è un design pattern – un intermediario, rappresentante di un altro oggetto, di cui ha la stessa interfaccia
    - un *proxy* [GoF] fornisce un surrogato o un segnaposto per un altro oggetto – per controllarne l'accesso
  - l'uso di stub e skeleton nasconde al programmatore il fatto che la comunicazione è distribuita
    - della comunicazione distribuita (e dei suoi dettagli) si occupano i proxy



Discussione

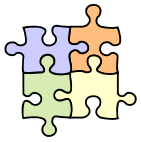
- Un aspetto fondamentale della tecnologia RPC (e successive) è la generazione automatica di stub e skeleton
  - basata su un opportuno linguaggio per la definizione di interfacce – *Interface Definition Language* (*IDL*)
  - realizzata da un "*compilatore d'interfacce*"





Discussione

- Uso – pragmatico e semplificato – di RPC
  - scrittura dell'*interfaccia del servizio* tramite l'IDL
    - l'interfaccia di un servizio remoto descrive: (i) quali sono le operazioni e (ii) quali sono i tipi usati dalle operazioni (per i parametri e i valori restituiti)
  - la *compilazione dell'interfaccia* produce sia codice lato client (stub) che codice lato server (skeleton) per le procedure del servizio
  - completamento del codice
    - lato server, vanno implementate le procedure del servizio – inoltre, nello skeleton – in opportuni segnaposti – vanno chiamate queste procedure che implementano il servizio
    - lato client, le chiamate alle procedure vanno rivolte allo stub
    - il programma server va mandato in esecuzione sul server – e il servizio va registrato presso il sistema operativo



Discussione

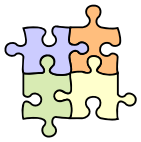
- Uso – pragmatico e semplificato –
  - scrittura d
    - l'int
      op
      pa
    - l
    - co
    - lato
    - il p
      e il servizio va registrato presso il sistema operativo

Ma la semantica di una chiamata di procedura remota è la stessa di una chiamata di procedura locale?

Se no, in cosa differisce?

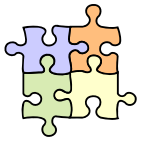
Ad esempio, può essere basata sul meccanismo noto della pila di attivazione – o richiede qualche estensione di questo meccanismo?

E il legame dei parametri?



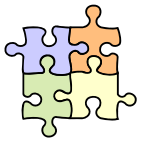
* Oggetti distribuiti

- Il *modello a oggetti distribuiti* fornisce un'astrazione di programmazione distribuita orientata agli oggetti
  - una collezione di oggetti distribuiti
    - gli oggetti possono risiedere in processi diversi
  - ciascun oggetto può
    - offrire servizi pubblici – tramite un'interfaccia
    - incapsulare uno stato privato
  - gli oggetti cooperano mediante l'invocazione di metodi
- **RMI** – *Remote Method Invocation*
  - un sistema RMI consente agli oggetti di un processo di invocare metodi di oggetti di altri processi
    - si tratta di un'evoluzione orientata agli oggetti di RPC
  - esiste anche un'accezione specifica – *Java RMI*



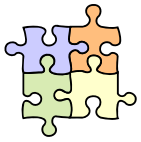
- Modello a oggetti – non distribuiti

- Ciascun *oggetto* incapsula stato e comportamento
- Gli oggetti risiedono solitamente in un singolo processo
- Ciascun oggetto implementa un'*interfaccia* (definita implicitamente o esplicitamente)
  - specifica dei metodi che possono essere invocati
- Un oggetto può essere usato conoscendone il *riferimento univoco*
  - ad es., usato in un'invocazione come destinatario

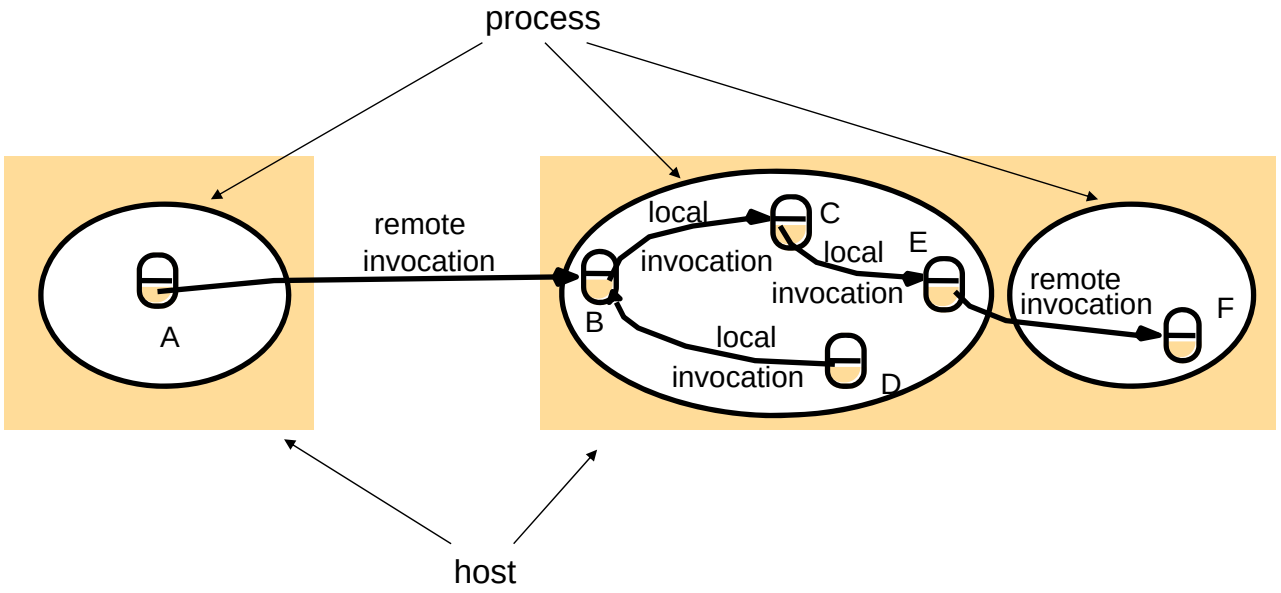


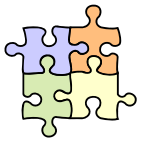
- Modello a oggetti distribuiti

- Ciascun *oggetto* incapsula stato e comportamento
- Oggetti locali e oggetti remoti
  - gli *oggetti locali* sono visibili localmente a un processo
  - gli *oggetti remoti* possono essere distribuiti in più processi/computer
- Ciascun oggetto remoto implementa un'*interfaccia remota* (definita esplicitamente)
  - specifica dei metodi che possono essere invocati remotamente
- Un oggetto remoto può essere usato conoscendone il *riferimento remoto univoco*
  - ad es., usato in un'*invocazione remota* come destinatario

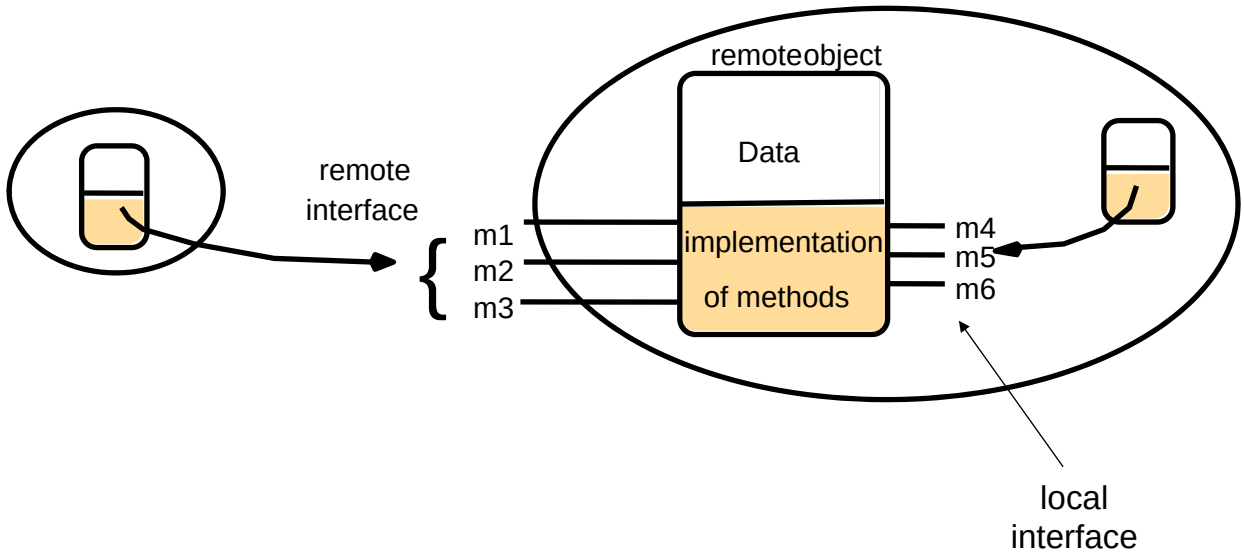


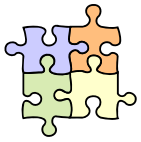
Oggetti e metodi remoti e locali



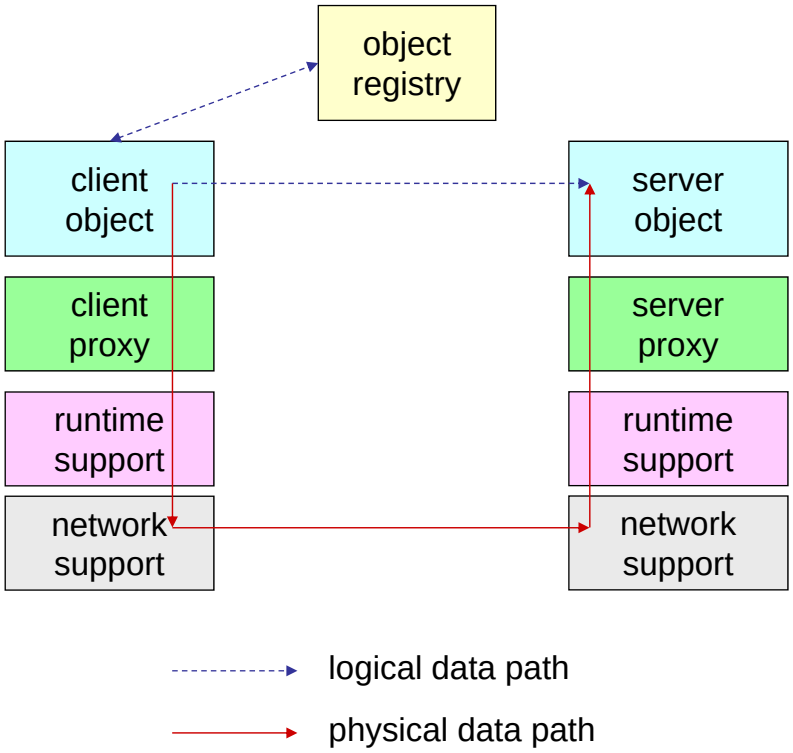


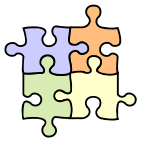
Oggetto remoto e sue interfacce





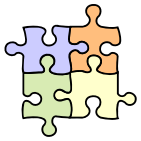
- Architettura di RMI





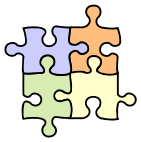
Dinamica di RMI

- Un processo (server) può esportare uno o più oggetti distribuiti
  - ciascun *oggetto distribuito*
    - è di un *tipo remoto* – definito da un'interfaccia remota
    - è identificato da un *riferimento remoto* – univoco in rete
- Gli oggetti distribuiti possono essere registrati presso un *object registry*
  - è un servizio distribuito che gestisce le corrispondenze tra identificatori simbolici e riferimenti remoti
- Un processo (client) può consultare l'object registry per ottenere un riferimento remoto ad un oggetto distribuito
  - in realtà ottiene un client proxy – che fornisce il supporto runtime che consente al client di interagire con l'oggetto remoto
- E la semantica delle chiamate remote? Sarà discussa più avanti



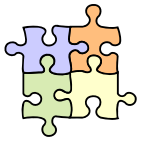
* Java RMI

- Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate
  - for a basic communication mechanism, the Java™ programming language supports sockets ...
  - however, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone
- An alternative to sockets is RPC, which abstracts the communication interface to the level of a procedure call ...
- RPC, however, does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed
  - in order to match the semantics of object invocation, distributed object systems require remote method invocation or RMI ...



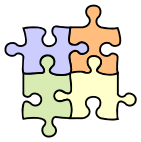
Java RMI

- The Java platform's remote method invocation system described in this specification has been specifically designed to operate in the Java application environment
  - the Java programming language's RMI system assumes the homogeneous environment of the Java virtual machine (JVM), and the system can therefore take advantage of the Java platform's object model whenever possible



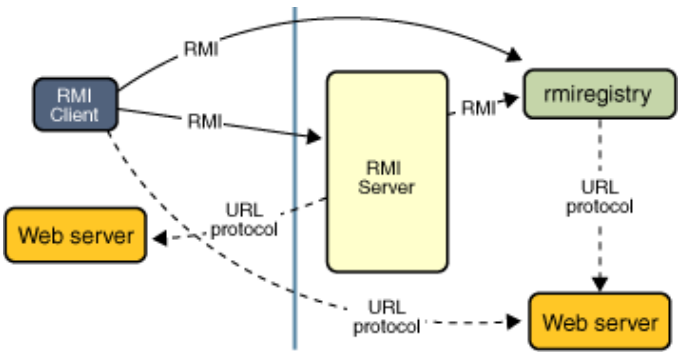
Java RMI

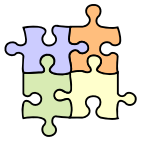
- The goals for supporting distributed objects in the Java programming language are
  - support seamless remote invocation on objects in different virtual machines
  - support callbacks from servers to applets
  - integrate the distributed object model into the Java programming language in a natural way while retaining most of the Java programming language's object semantics
  - make differences between the distributed object model and local Java platform's object model apparent
  - make writing reliable distributed applications as simple as possible
  - preserve the type-safety provided by the Java platform's runtime environment
  - support various reference semantics for remote objects; for example live (nonpersistent) references, persistent references, and lazy activation
  - maintain the safe environment of the Java platform provided by security managers and class loaders
- Underlying all these goals is a general requirement that the RMI model be both simple (easy to use) and natural (fits well in the language)



Java RMI

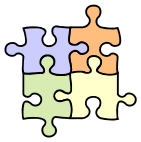
- *Java Remote Method Invocation* (*Java RMI*) consente a un oggetto in esecuzione in una macchina virtuale Java di invocare metodi di oggetti in esecuzione in altre macchine virtuali Java
  - Java RMI fornisce un meccanismo per la comunicazione remota tra programmi Java





- L'applicazione Hello

- Java RMI viene ora presentato con un semplice esempio
  - un oggetto remoto (ovvero, distribuito) implementa un servizio ***Hello*** per calcolare un saluto personalizzato
  - in una prima versione, consideriamo interfaccia remota, servant, server e client
    - senza però distinguere tra "componenti" e "connettori"
  - sarà mostrata più avanti una seconda versione della stessa applicazione – basata su una distinzione più netta tra "componenti" e "connettori"

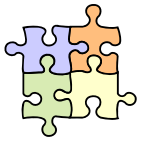


- Interfaccia remota del servizio

```
package asw.asw830.service;

import java.rmi.*;

public interface Hello extends java.rmi.Remote {
    public String sayHello(String name)
            throws HelloException, java.rmi.RemoteException;
}
```

in **rosso** indichiamo il codice che dipende da Java RMI

- In Java RMI, un oggetto/tipo è considerato remoto/distribuito se implementa un'*interfaccia remota*
  - ovvero, un'interfaccia che estende l'interfaccia ***java.rmi.Remote***
    - si tratta di un'interfaccia marker, che non dichiara metodi
  - inoltre, i metodi remoti devono dichiarare di sollevare l'eccezione ***java.rmi.RemoteException*** – ma il servente non deve sollevare eccezioni di questo tipo

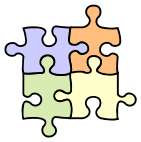


Eccezioni del servizio

- Nella definizione di un servizio remoto RMI è possibile identificare due tipi di eccezioni
  - eccezioni legate al servizio, di natura “funzionale”

```
package asw.asw830.service;

public class HelloException extends Exception {
        public HelloException(String message) {
                super(message);
        }
}
```

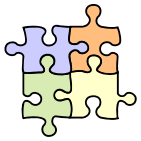
  - l’eccezione ***java.rmi.RemoteException***, legata alla natura distribuita del servizio
    - ogni servizio distribuito deve sollevare eccezioni legate alla natura distribuita del servizio – idealmente, però, indipendenti dalla specifica tecnologia con cui il servizio distribuito è realizzato – in pratica, come in questo caso, queste eccezioni dipendono dalla tecnologia usata



- Servente – implementazione del servizio

```
package asw.asw830.service.impl;

import asw.asw830.service.*;

public class HelloServant implements Hello {
    public HelloServant() {
        super();
    }
    public String sayHello(String name) throws HelloException {
        if (name==null) {
            throws new HelloException("name is null");
        } else {
            return "Hello, " + name + "!";
        }
    }
}
```

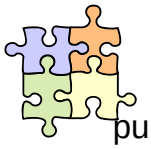
- Un *servente* (servant) è un’istanza di una classe che implementa un’interfaccia remota
  - notare che l’implementazione del servizio non solleva mai ***java.rmi.RemoteException***



- Server

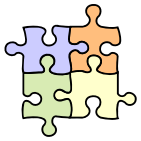
- Il *server* per un oggetto remoto è un'applicazione che
  - istanzia l'oggetto servente – e lo esporta come oggetto remoto, di tipo ***java.rmi.server.UnicastRemoteObject***
  - registra questo oggetto remoto presso l'*object registry* – associandogli un nome simbolico

```
package asw.asw830.server;

import asw.asw830.service.*;
import asw.asw830.service.impl.*;

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

public class HelloServer {
    public static void main(String[] args) {
        ... crea il servente ...
        ... esporta il servente come oggetto RMI ...
        ... registra il servente presso il registry RMI ...
    }
}
```

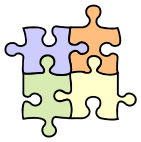
Server

```
public static void main(String[] args) {
    /* crea il servente */
    Hello hello = new HelloServant();
    /* esporta il servente come oggetto RMI, e lo registra sul registry RMI */
    /* crea il security manager */
    if (System.getSecurityManager()==null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        /* esporta il servente come oggetto RMI */
        Hello helloStub = (Hello) UnicastRemoteObject.exportObject(hello, 0);
        /* registra il servente presso il registry RMI */
        String helloServiceName = "rmi:/asw830/Hello";
        String registryHost = "192.168.50.101";
        Registry registry = LocateRegistry.getRegistry(registryHost);
        registry.rebind(helloServiceName, helloStub);
        /* servente registrato */
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



Server

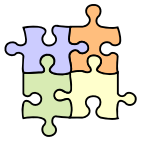
- Osservazioni
  - il security manager ha lo scopo di proteggere l'accesso a risorse da parte di codice scaricato da altri siti
    - è necessario anche definire un file di policy – server.policy
  - il metodo ***UnicastRemoteObject.exportObject*** esporta l'oggetto ***hello*** – che ha un'interfaccia remota – in modo tale che possa effettivamente ricevere invocazioni remote
    - in pratica, lo collega a una porta TCP
    - altrimenti, il servente può estendere ***UnicastRemoteObject***
  - il metodo ***LocateRegistry.getRegistry(registryHost)*** ha lo scopo di ottenere un riferimento al registry RMI
    - in realtà, un remote proxy al registry RMI
  - il metodo ***Registry.rebind*** registra l'oggetto remoto sul registry
    - associa all'oggetto remoto un nome simbolico – il client deve conoscerlo per poter accedere all'oggetto remoto



- Client

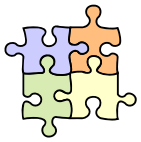
- Un *client* che vuole usare un oggetto remoto può
  - ottenere un riferimento all'oggetto remoto tramite l'object registry – conoscendo il nome simbolico dell'oggetto remoto
    - in realtà otterrà un remote proxy all'oggetto distribuito
  - usare l'oggetto remoto – tramite questo remote proxy

```
package asw.asw830.client;

import asw.asw830.service.*;

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

public class HelloClient {
    public static void main(String args[]) {
        ... cerca il servizio remoto ...
        ... utilizza il servizio remoto ...
    }
}
```



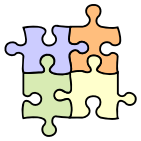
Client

```
/* cerca il servizio remoto */
Hello hello = null;
/* crea il security manager */
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}
try {
   /* ottiene un proxy al servizio remoto */
    String helloServiceName = "rmi:/asw830/Hello";
    String registryHost = "192.168.50.101";
    Registry registry = LocateRegistry.getRegistry(registryHost);
    hello = (Hello) registry.lookup(helloServiceName);
} catch (RemoteException e) {
    ... getRegistry e lookup possono sollevare RemoteException ...
} catch (NotBoundException e) {
     ... registry.lookup può sollevare NotBoundException ...
}
```

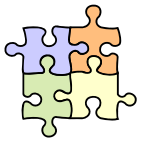
Client

```
/* usa il servizio remoto hello */
try {
    String name = "Luca";
    String result = hello.sayHello(name);
    ... fai qualcosa con result ...
} catch (HelloException e) {
    ... l'uso di hello può sollevare HelloException ...
} catch (java.rmi.RemoteException e) {
    ... RMI può sollevare java.rmi.RemoteException ...
}
```



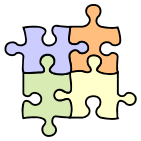
Client

- Osservazioni
  - anche in questo caso, l'uso del security manager richiede anche la definizione di un file di policy – client.policy
    - importante soprattutto per il client, se deve eseguire codice scaricato da un server
  - il metodo ***LocateRegistry.getRegistry*** ha lo scopo di ottenere il (remote proxy al) registry
  - il metodo ***Registry.lookup*** ha lo scopo di ottenere un riferimento remoto all'oggetto remoto ***Hello***
    - in realtà, restituisce un remote proxy – che incapsula un riferimento remoto all'oggetto remoto ***Hello***



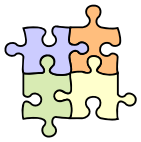
Java RMI, in pratica (1)

- Compilazione ed esecuzione di un'applicazione Java RMI sono (in parte) diverse rispetto a quelle di un'applicazione Java
  - qui illustrato con riferimento alla piattaforma Windows
- Compilazione – separatamente
  - dell'interfaccia remota
  - del server – con la definizione dell'interfaccia del servizio nel "build path"
  - del client – con la definizione dell'interfaccia del servizio nel "build path"
  - nota: con Java 5, era necessario usare il comando *rmic* per generare il proxy lato client dall'interfaccia remota
    - con Java 6/7/8, il proxy lato client e quello lato server sono "virtuali" – ottenuti dall'interfaccia remota sulla base di meccanismi di riflessione

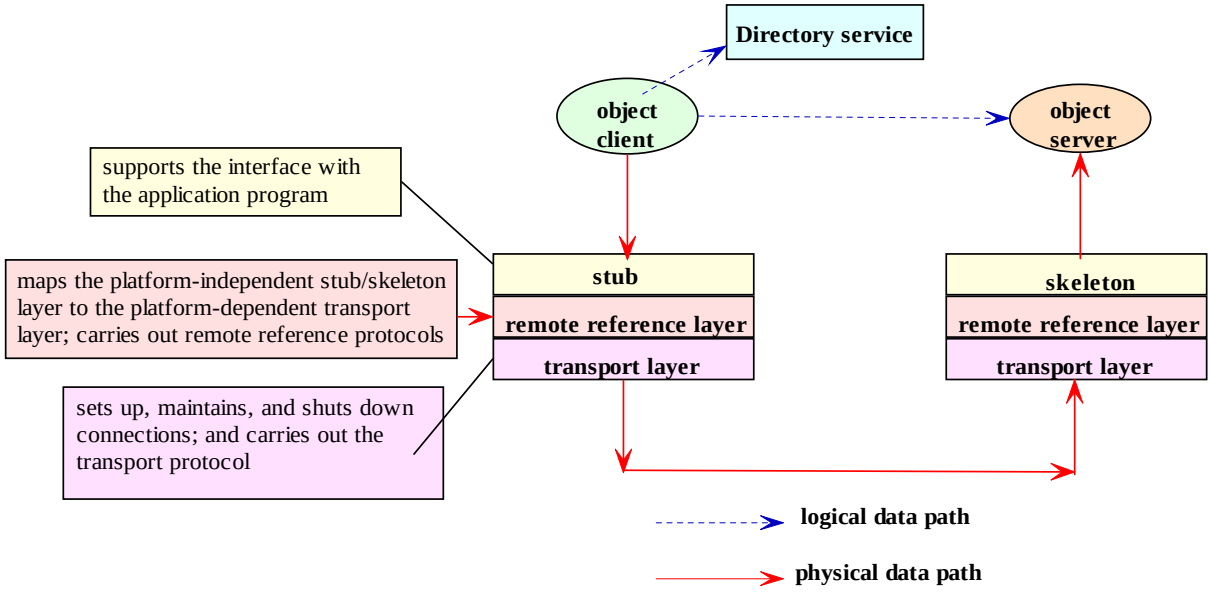


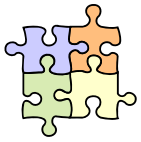
Java RMI, in pratica (2)

- Avvio dell'RMI registry – sull'host incaricato di questo compito – nel nostro esempio, corrisponde all'host che ospita il server
  - ***set CLASSPATH=***
  - ***start rmiregistry***
- Avvio del server
  - con uno script che avvia ***HelloServer***
  - ma specificando anche alcuni parametri relativi all'uso di RMI e del security manager
- Avvio del client
  - con uno script che avvia ***HelloClient***
  - ma specificando anche alcuni parametri relativi all'uso di RMI e del security manager

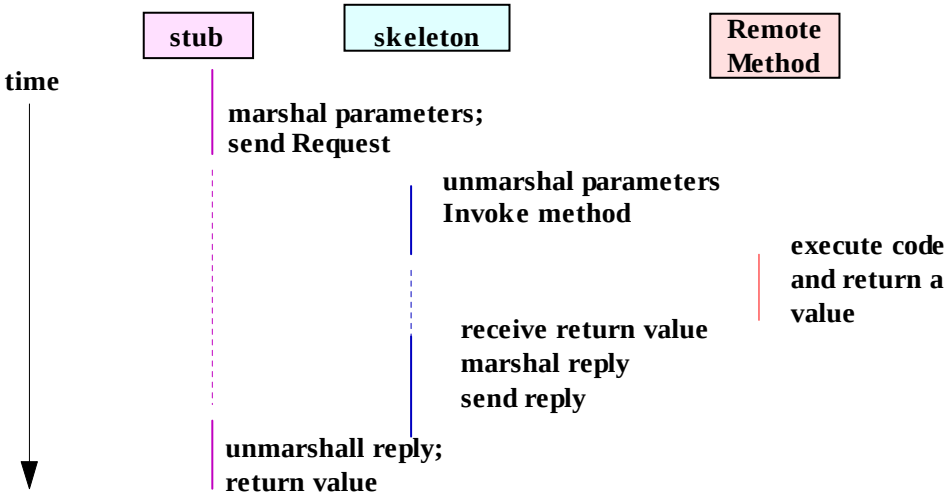


* Architettura di Java RMI

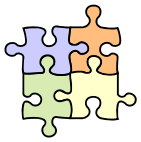




Interazione tra client e server

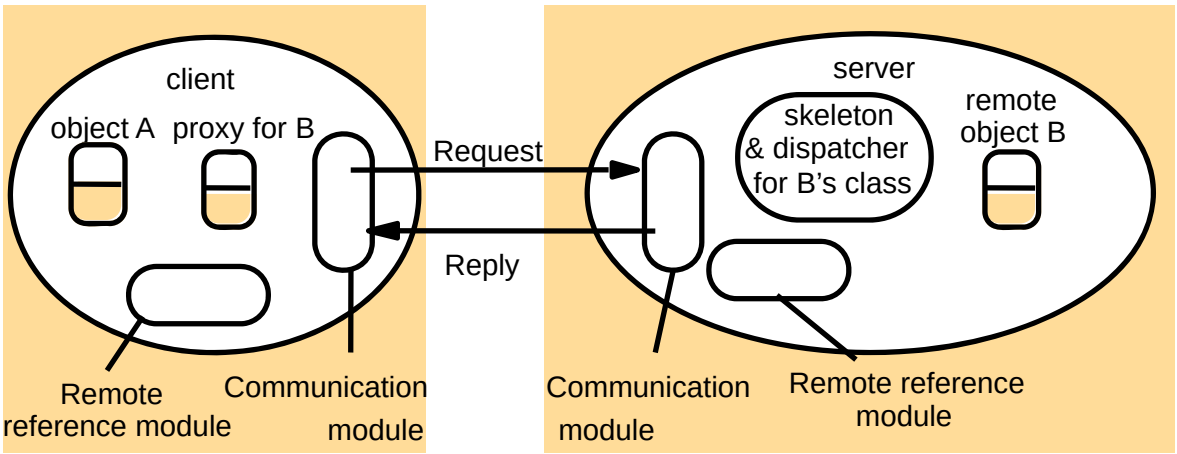


(based on http://java.sun.com.marketing/collateral/javarim.html)

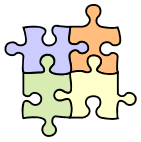


- Implementazione di RMI

- L'implementazione di RMI richiede diversi oggetti e moduli

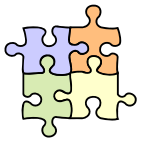


  - in alcuni casi si tratta di moduli standard
  - altri sono generati automaticamente – da ***rmic*** – o sono virtuali – sulla base di meccanismi di riflessione



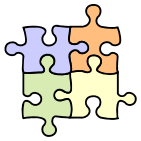
Implementazione di RMI (1)

- Proxy (proxy lato client)
  - rappresentante locale di un oggetto remoto
    - tramite il proxy, l'oggetto client comunica logicamente con l'oggetto server
  - implementa l'interfaccia remota
    - per adattare e inoltrare le richieste all'oggetto remoto e gestire le sue risposte
  - ha lo scopo di nascondere gli aspetti della comunicazione fisica
    - ad es., la posizione dell'oggetto server (viene visto dal client mediante il proxy), marshalling dei dati e unmarshalling dei risultati, invio e ricezione di messaggi
  - delega gran parte del lavoro al modulo di comunicazione



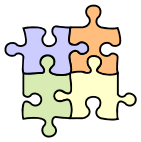
Implementazione di RMI (2)

- Moduli di comunicazione
  - il protocollo richiesta-risposta viene eseguito da una coppia di moduli di comunicazione cooperanti
  - trasmettono messaggi richiesta e risposta
  - i due moduli di comunicazione hanno un ruolo fondamentale nella definizione della *semantica per l'invocazione di metodi remoti* – discussa più avanti
    - in particolare, nel caso di Java RMI, questi moduli implementano una semantica "at most once" (descritta dopo)
- Modulo dei riferimenti remoti
  - responsabile delle corrispondenze tra riferimenti locali e remoti – e della creazione di riferimenti remoti
  - tavola degli oggetti remoti – elenco degli oggetti remoti offerti – elenco dei proxy conosciuti



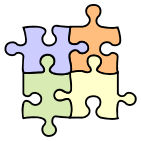
Implementazione di RMI (3)

- Dispatcher
  - il server ha un dispatcher per ogni classe di oggetto remoto
  - il dispatcher riceve messaggi richiesta dal modulo di comunicazione – seleziona l'operazione da eseguire – ne delega la gestione allo skeleton
- Skeleton (proxy lato server)
  - proxy del client nel lato server
  - il server ha uno skeleton per ogni classe di oggetto remoto
  - implementa metodi corrispondenti a quelli dell'interfaccia remota
  - ha lo scopo di estrarre i parametri dal messaggio richiesta e invocare il metodo corrispondente nell'oggetto remoto
  - aspetta l'esecuzione del metodo, quindi costruisce il messaggio risposta dal suo risultato



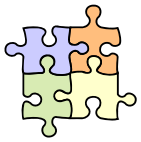
Implementazione di RMI (4)

- Compilatore di interfacce
  - responsabile della generazione delle classi per i proxy – sia per lo stub che per dispatcher e skeleton – a partire dall'interfaccia del servizio
  - in realtà, con Java 6/7/8 l'uso del compilatore di interfacce non è strettamente necessario
    - vengono utilizzati dei proxy "generici" – sia lato server che lato client – basati sull'applicazione di meccanismi di riflessione alla definizione dell'interfaccia remota
- Binder
  - consente di ottenere riferimenti a oggetti remoti
  - gestisce una corrispondenza tra nomi simbolici (stringhe) e riferimenti remoti



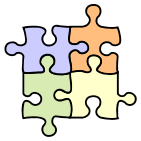
* Confronto tra RMI e socket

- Alcune differenze distintive di RMI e socket
  - i socket lavorano più vicini al sistema operativo – l'overhead a runtime è minore
    - RMI richiede del software di supporto addizionale
    - RMI richiede un certo overhead a runtime
  - l'astrazione offerta da RMI facilita lo sviluppo del software
    - programmi più facili da comprendere, mantenere e verificare
  - i socket sono indipendenti dal linguaggio e dalla piattaforma
    - Java RMI è chiaramente dipendente dal linguaggio – ma è indipendente dalla piattaforma
    - esistono altre implementazione di RMI che sono anche indipendenti dal linguaggio – ad es., CORBA



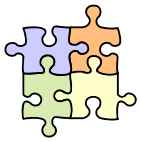
* Java RMI e connettori

- Nell'esempio mostrato
  - la realizzazione degli aspetti funzionali del servizio e del client è intrecciata con gli aspetti relativi all'utilizzo di Java RMI
- Chiaramente, Java RMI può essere usato nella scrittura di codice in cui c'è una separazione più netta tra
  - componenti – ovvero, logica applicativa – che devono essere (per quanto possibile) indipendenti dalle tecnologie usate
  - connettori – che possono e devono dipendere dalle particolari tecnologie usate
- Ma è davvero utile ragionare, in questo caso, sulla separazione tra codice componente e codice connettore?
  - si tratta in buona misura di un problema stilistico
  - tuttavia, può essere importante ragionare esplicitamente sulle interfacce remote – e sulla gestione delle eccezioni remote



Indipendenza da RMI

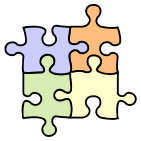
- Un obiettivo che si potrebbe voler raggiungere è l'indipendenza del client e del servente (che sono componenti) da RMI (che è una tecnologia)
  - è possibile incapsulare la dipendenza da RMI nel connettore (lato client e lato server)
- È un problema di "adattamento" – può essere risolto usando il design pattern GoF ***Adapter***
  - ***Adapter*** consente di convertire l'interfaccia di una classe in un'altra interfaccia richiesta dal client
  - ***Adapter*** consente a classi diverse di operare insieme quando ciò non sarebbe altrimenti possibile a causa di interfacce incompatibili [GoF]
- Attenzione, nel nostro caso dobbiamo effettuare una doppia conversione



Interfacce remote

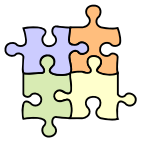
- Nella realizzazione di servizi da fruire in modo distribuito, è inevitabile dover prendere in considerazione il fatto che un servizio possa essere non disponibile
  - in alcuni casi è utile rappresentare tale possibilità in modo indipendente dalla specifica tecnologia usata per la comunicazione distribuita
- Che cosa possiamo dire dell'interfaccia del nostro servizio? È indipendente dalla tecnologia usata?

```
package asw.asw830.service;

import java.rmi.*;

public interface Hello extends java.rmi.Remote {
    public String sayHello(String name)
            throws HelloException, java.rmi.RemoteException;
}
```



Pattern Convert Exceptions

- Il design pattern *Convert Exceptions*
  - nell'ambito di un componente o sottosistema, evita di sollevare eccezioni di livello più basso che derivano da un componente o sottosistema di livello più basso
  - piuttosto, prova a gestire le eccezioni di livello più basso – oppure, converti l'eccezione di livello più basso in una nuova eccezione che è significativa al livello di questo componente o sottosistema
    - l'eccezione di livello più alto di solito incapsula quella di livello più basso, aggiunge informazioni, per rendere l'eccezione più significativa/utile dove sarà catturata
  - ad es., in un sottosistema per la persistenza
    - non sollevare *SQLException*
    - piuttosto, solleva *PersistenceException* oppure *EmployeeNotFoundException*

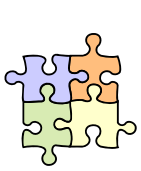


- Interfaccia del servizio

```
package asw.asw830.service;

public interface Hello {
        public String sayHello(String name)
                throws HelloException, RemoteException;
}
```

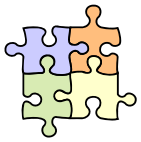
```
package asw.asw830.service;

/* HelloException è un'eccezione funzionale, legata al servizio. */
public class HelloException extends Exception {
        public HelloException(Exception cause) {
                super(cause);
        }
}
```

```
package asw.asw830.service;

/* asw.asw830.service.RemoteException indica un problema nell'accesso
 * remoto al servizio – in modo indipendente dalla tecnologia usata */
public class RemoteException extends Exception {
        public RemoteException(Exception cause) {
                super(cause);
        }
}
```



- Implementazione del servizio

```
package asw.asw830.service.impl;

import asw.asw830.service.*;

public class HelloServant implements Hello {

    public HelloServant() {
        super();
    }

    public String sayHello(String name) throws HelloException {
        if (name==null)
            throws new HelloException("name is null");
        else
            return "Hello, " + name + "!";
    }

}
```

- Interfaccia remota del servizio

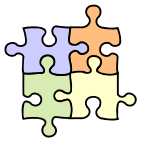
```
package asw.asw830.service;

public interface Hello {
    public String sayHello(String name)
        throws HelloException, RemoteException;
}
```

```
package asw.asw830.service.rmi;

import asw.asw830.service.*;

import java.rmi.*;

public interface HelloRmiInterface extends java.rmi.Remote {
    public String sayHello(String name)
        throws HelloException, java.rmi.RemoteException,;
}
```

- Le interfacce ***Hello*** e ***HelloRmiInterface*** sono tra loro *simili*
  - si noti però che non esiste nessuna dipendenza “di compilazione” esplicita
  - anche ***asw.asw830.service.RemoteException*** e ***java.rmi.RemoteException*** sono solo simili

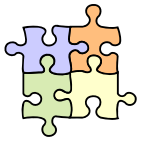
- Proxy lato server

```
package asw.asw830.server;

import asw.asw830.service.*;
import asw.asw830.service.rmi.*;

/* Proxy lato server per Hello, che è un servente RMI. */
public class HelloServerProxy implements HelloRmiInterface {

    /* il vero servizio hello */
    private Hello hello;

    public HelloServerProxy(Hello hello) {
        super();
        this.hello = hello;
    }

    ... segue ...
```

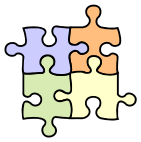
- Proxy lato server

```
    public String sayHello(String name)
            throws HelloException, java.rmi.RemoteException {
        try {
            return hello.sayHello(name);
        } catch (HelloException e) {
            /* rilancia e */
            throw e;
        } catch (asw.asw830.service.RemoteException e) {
            /* converte l'eccezione remota (indipendente dalla tecnologia)
             * in un'eccezione remota (RMI) */
            throw new java.rmi.RemoteException(e);
        }
    }

}
```



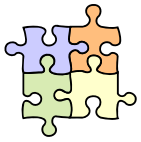
- Server

```
package asw.asw830.server;

import asw.asw830.service.*;
import asw.asw830.service.impl.*;
import asw.asw830.service.rmi.*;

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

public class HelloServer {

    public static void main(String[] args) {
        ... crea il servente RMI ...
        ... esporta il servente come oggetto RMI ...
        ... registra il servente presso il registry RMI...
    }

}
```

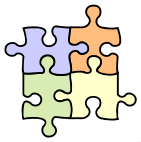
- Server

```
/* crea il vero servente */
Hello hello = new HelloServant();
/* esporta il servente come oggetto RMI, e lo registra sul registry */
/* crea il security manager */
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}
try {
    /* crea il servente RMI (è un proxy al vero servente) */
    HelloRmiInterface helloService = new HelloServerProxy(hello);
    /* esporta il servente RMI */
    HelloRmiInterface helloStub =
        (HelloRmiInterface) UnicastRemoteObject.exportObject(helloService, 0);
    /* registra il servente presso il registry RMI*/
    String helloServiceName = "rmi:/asw830/Hello-cc";
    String registryHost = "192.168.50.101";
    Registry registry = LocateRegistry.getRegistry(registryHost);
    registry.rebind(helloServiceName, helloStub);
    /* servente registrato */
} catch (Exception e) {
    ... gestisci e ...
}
```



- Client

```
package asw.asw830.client;

import asw.asw830.service.*;

/* client del servizio hello */
public class HelloClient {

    private Hello hello;

    public HelloClient(Hello hello) {
        this.hello = hello;
    }

    private void run() {
      try {
          ...
          ... hello.sayHello("Luca") ...
          ...
      } catch (HelloException e) {
          ... gestisci e ...
      } catch (RemoteException e) {
          ... gestisci e ...      }
   }

}
```

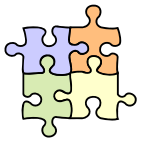



- Main

```
package asw.asw830.client;

import asw.asw830.service.*;
import asw.asw830.client.connector.*;

/* Applicazione client: ottiene e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto HelloClient. */
    public static void main(String[] args) {
        Hello hello = ServiceFactory.getInstance().getHello();
        /* crea il client e gli inietta il servizio da cui dipende */
        HelloClient client = new HelloClient(hello);
        client.run();
    }

}
```

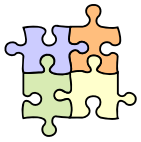
- Factory per il proxy lato client

```
package asw.asw830.client.connector;

import asw.asw830.service.*;

/* Factory per il servizio Hello. */
public class ServiceFactory {
    ... variabile e metodo per singleton ...

    /* Factory method per il servizio hello. */
    public Hello getHello() {
        Hello service = null;
        try {
            String helloServiceName = "rmi:/asw830/Hello";
            String registryHost = "192.168.50.101";
            service = new HelloClientProxy(helloServiceName, registryHost);
        } catch(Exception e) {
            ... gestisci e ...
        }
        return service;
    }
}
```

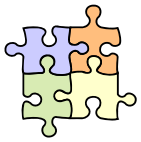
- Proxy lato client (1)

```
package asw.asw830.client.connector;

import asw.asw830.service.*;
import asw.asw830.service.rmi.*;

import java.rmi.registry.*;

public class HelloClientProxy implements Hello {

    /* riferimento al (proxy al) servizio remoto RMI */
    private HelloRmiInterface hello;

    public HelloClientProxy(String helloUrl, String registryName) { ... }

    public String sayHello(String name)
        throws HelloException, asw.asw830.service.RemoteException { ... }

}
```



Proxy lato client (2)

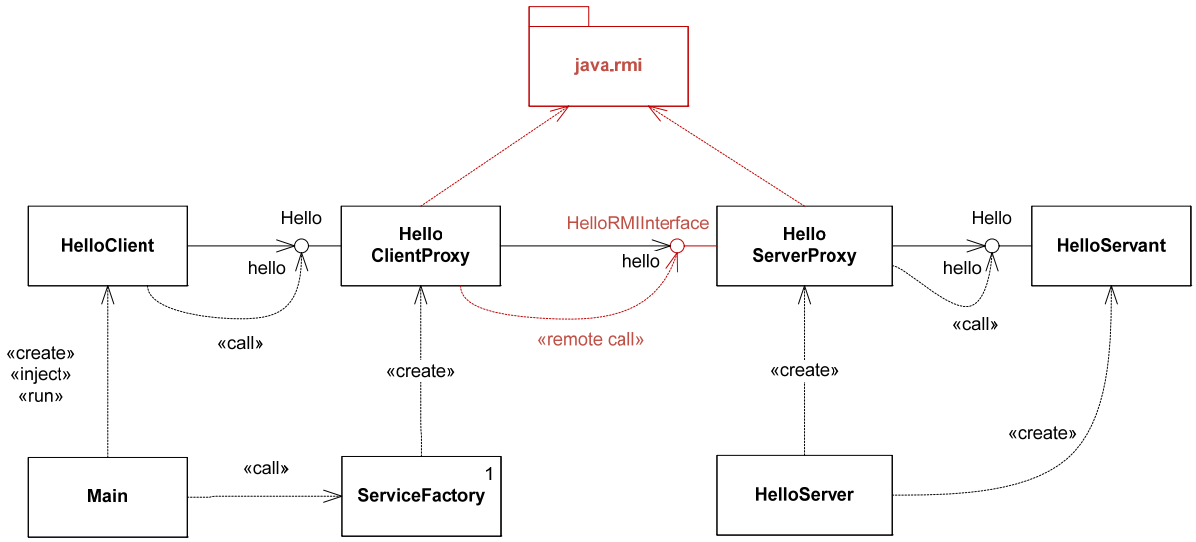
```
public HelloClientProxy(String registryHost, String helloServiceName) {
    /* crea il security manager */
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        /* cerca un riferimento al servizio remoto */
        Registry registry = LocateRegistry.getRegistry(registryHost);
        this.hello = (HelloRmiInterface) registry.lookup(helloServiceName);
    } catch (Exception e) {
        ... gestisci e ...
    }
}
```

Proxy lato client (3)

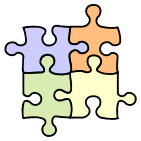
```
public String sayHello(String name)
        throws HelloException, asw.asw830.service.RemoteException {
    try {
        return hello.sayHello(name);
    } catch (HelloException e) {
        /* rilancia e */
        throw e;
    } catch (java.rmi.RemoteException e) {
        /* converte l'eccezione remota */
        throw new asw.asw830.service.RemoteException(e.getMessage());
    }
}
```

In sintesi



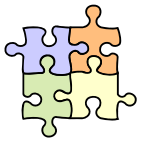
- Discussione

- Questo “esercizio di stile” ha mostrato che, in linea di principio, è sempre possibile
  - identificare una porzione di codice “componente” e una porzione di codice “connettore”
  - separando così – in buona misura – il codice “componente” dal codice “connettore”
- Questo esercizio ha mostrato che è inoltre possibile rendere i componenti indipendenti da scelte tecnologiche circa i connettori
- Nel seguito del corso questo “esercizio di stile” non sarà più fatto
  - verrà esemplificato l’uso di alcune tecnologie e alcuni strumenti di middleware
  - non ci si preoccuperà però troppo di questa separazione tra codice “componente” e codice “connettore” – o di realizzare l’indipendenza dalle tecnologie



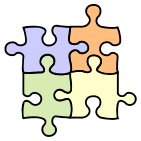
* Ulteriori aspetti

- Vengono ora discussi ulteriori aspetti relativi a RMI (in genere) oppure a Java RMI (in particolare)



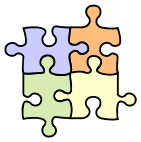
- Semantica dell'invocazione remota

- Nei sistemi distribuiti si possono verificare diversi fallimenti nella comunicazione – in particolare
  - messaggi persi o tramessi male
  - messaggi trasmessi più volte
  - messaggi trasmessi fuori ordine
- In particolare, nella realizzazione di un protocollo richiesta-risposta, bisogna considerare la possibilità di perdita del messaggio di richiesta o del messaggio di risposta
  - per la gestione di queste situazioni sono possibili diverse opzioni
    - ripetizione del messaggio di richiesta – lato client
    - filtraggio di richieste duplicate – lato server
    - ritrasmissione di risultati – lato server



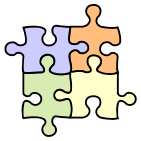
Semantica dell'invocazione remota

- Il protocollo richiesta-risposta può essere implementato in modi diversi – in particolare, usando (o meno) le seguenti opzioni
  - ripetizione del messaggio di richiesta – lato client
    - fino a quando non arriva una risposta – o comunque per un numero massimo di volte
  - filtraggio di richieste duplicate – lato server
    - tenendo una storia delle richieste
  - ritrasmissione di risultati – lato server
    - tenendo una storia delle richieste-risposte
- L'uso combinato di opzioni diverse fornisce un ventaglio di semantiche differenti per l'invocazione di metodi remoti
  - maybe – non ripetere richieste
  - at least once – ritrasmetti richieste, non filtrare duplicati
  - at most once – ritrasmetti richieste, filtra duplicati, ritrasmetti risposte



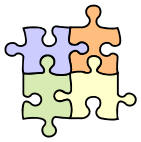
Semantica dell'invocazione remota (1)

- Possibili semantiche per l'invocazione di metodi remoti
  - *maybe* – non ripetere richieste
    - un'invocazione può terminare con
      - la restituzione di un risultato – vuol dire che il metodo è stata eseguito una sola volta
      - oppure, la segnalazione di un'eccezione remota – il metodo potrebbe non essere stato mai eseguito, ma potrebbe anche essere stato eseguito (ma solo una volta)
    - è un'opzione per Corba
  - *at least once (or maybe not)* – ritrasmetti richieste, non filtrare duplicati, ri-esegui il metodo
  - *at most once* – ritrasmetti richieste, filtra duplicati, non rieseguire il metodo, ritrasmetti risposte
- *La semantica dell'invocazione di metodi locali è exactly once*



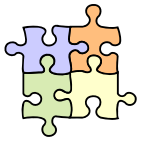
Semantica dell'invocazione remota (2)

- Possibili semantiche per l'invocazione di metodi remoti
  - *maybe* – non ripetere richieste
  - *at least once (or maybe not)* – ritrasmetti richieste, non filtrare duplicati, ri-esegui il metodo
    - un'invocazione può terminare con
      - la restituzione di un risultato – vuol dire che il metodo è stata eseguito una o anche più volte
      - oppure, la segnalazione di un'eccezione remota – il metodo potrebbe non essere stato mai eseguito, ma potrebbe anche essere stato eseguito una o più volte
  - *at most once* – ritrasmetti richieste, filtra duplicati, non rieseguire il metodo, ritrasmetti risposte
- La semantica dell'invocazione di metodi locali è exactly once



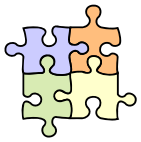
Semantica dell'invocazione remota (3)

- Possibili semantiche per l'invocazione di metodi remoti
  - *maybe* – non ripetere richieste
  - *at least once (or maybe not)* – ritrasmetti richieste, non filtrare duplicati, ri-esegui il metodo
  - *at most once* – ritrasmetti richieste, filtra duplicati, non rieseguire il metodo, ritrasmetti risposte
    - un'invocazione può terminare con
      - la restituzione di un risultato – vuol dire che il metodo è stata eseguito esattamente una volta
      - oppure, la segnalazione di un'eccezione remota – il metodo potrebbe non essere mai stato eseguito, oppure potrebbe essere stato eseguito (ma solo una volta)
    - è la semantica di default per Java RMI e Corba
- La semantica dell'invocazione di metodi locali è exactly once



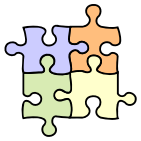
Semantica dell'invocazione remota (4)

- Possibili semantiche per l'invocazione di metodi remoti
  - *maybe* – non ripetere richieste
  - *at least once (or maybe not)* – ritrasmetti richieste, non filtrare duplicati, ri-esegui il metodo
  - *at most once* – ritrasmetti richieste, filtra duplicati, non rieseguire il metodo, ritrasmetti risposte
- La semantica dell'invocazione di metodi locali è *exactly once*
  - nessuna "eccezione remota" è possibile
  - un'invocazione può terminare solo con la restituzione di un risultato
    - vuol dire che il metodo viene eseguito esattamente una volta
- Si noti che nessuna delle tre semantiche per l'invocazione di metodi remoti corrisponde ad *exactly once*



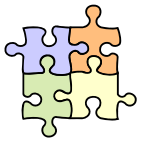
Discussione

- Dunque, ci sono tre possibili semantiche per l'invocazione di metodi remoti
  - differiscono tra loro per ciò che è in grado di capire il client di un'operazione quando
    - l'operazione termina e restituisce un risultato (o solleva un'eccezione funzionale)
    - viene segnalata un'eccezione remota
  - nessuna di queste tre semantiche corrisponde alla semantica *exactly once* dell'invocazione di metodi locali



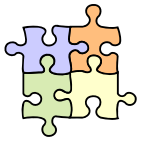
- Concorrenza

- Le esecuzioni dei metodi remoti avvengono in modo concorrente
  - un oggetto (server) distribuito può essere condiviso da più client
  - un oggetto (server) distribuito vive in un singolo processo
    - si tratta di un unico oggetto individuale nello heap per quel processo
  - ciascuna diversa invocazione remota viene eseguita nell'ambito di un thread (lato server) differente e separato
    - dunque è possibile che uno stesso oggetto sia incaricato dell'esecuzione concorrente di più metodi
  - lo stato degli oggetti condivisi va gestito in modo opportuno
  - può essere utile (ma talvolta è invece dannoso) dichiarare *synchronized* i metodi remoti
    - i metodi sincronizzati di uno stesso oggetti vengono eseguiti in modo mutuamente esclusivo da thread separati



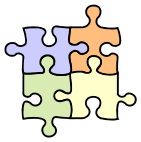
- Oggetti remoti con stato

- Un oggetto remoto può avere variabili d'istanza – che rappresentano lo stato di quell'oggetto
  - un oggetto remoto registrato presso l'object registry è visibile – e dunque condiviso – da tutti i suoi client
  - lo stato di un oggetto remoto condiviso può mantenere informazioni condivise da tutti i suoi client
  - attenzione, non stiamo parlando di stato conversazionale



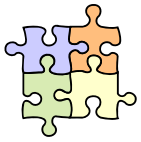
Esempio: Counter – interfaccia

```
package asw.asw830.counter;

import java.rmi.*;

public interface Counter extends Remote {
        /* Incrementa e restituisce il valore del contatore. */
        public int getCounter() throws RemoteException;
}
```

Esempio: Counter – servente

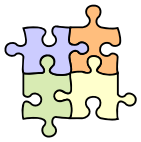
```
package asw.asw830.counter.implr;

import asw.asw830.counter.*;

public class CounterImpl implements Counter {

        private int counter;    // il contatore

        /* Crea un nuovo contatore, inizialmente nullo. */
        public CounterImpl() {
                super();
                this.counter = 0;
        }

        /* Incrementa e restituisce il valore del contatore. */
        public synchronized int getCounter() {
                this.counter++;
                return this.counter;
        }
}
```



Esempio: Counter – servente

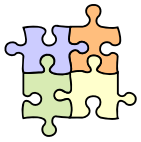
```
package asw.asw830.counter.implr;

import asw.asw8
public c
        /* Crea
        public CounterImpl() {
                super();
                this.count = 0;
        }

        /* Incrementa e restituisce il valore del contatore. */
        public synchronized int getCounter() {
                this.counter++;
                return this.counter;
        }
}
```

Questo metodo deve essere dichiarato synchronized.
Infatti, l'unica istanza di questa classe potrà essere utilizzata da molti client.



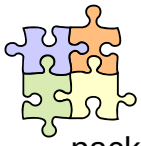
Esempio: Counter – server (1)

```
package asw.asw830.counter.server;

import asw.asw830.counter.*;
import asw.asw830.counter.impl.*;

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

public class CounterServer {

    public static void main(String[] args) {
        ... crea il servente ...
        ... registra il servente ...
    }
}
```



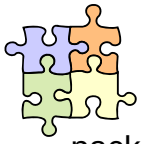
Esempio: Counter – server (2)

```
public static void main(String[] args) {
    /* crea il servente */
    Counter counterService = new CounterImpl();

    ... crea il security manager ...

    /* esporta e registra il servente RMI */
    try {
        Counter counterStub =
            (Counter) UnicastRemoteObject.exportObject(counterService, 0);
        String counterServiceName = "rmi:/asw830/Counter";
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(counterServiceName, counterStub);
        /* servente registrato */
    } catch (Exception e) {
        ... gestisci e ...
    }
}
```

Esempio: Counter – client (main)

```
package asw.asw830.counter.client;

import asw.asw830.counter.Counter;
import asw.asw830.counter.client.connector.ServiceFactory;

/* Applicazione client: ottiene e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto CounterClient. */
    public static void main(String[] args) {
        Counter counter = ServiceFactory.getInstance().getCounter();
        CounterClient client = new CounterClient(counter);
        client.run();
    }

}
```



Esempio: Counter – client (service factory)

```
package asw.asw830.counter.client.connector;

import asw.asw830.counter.Counter;

import java.rmi.*;

/* Factory per il servizio Counter. */
public class ServiceFactory {

    ... variabile, costruttore e metodo per singleton ...

    /* Factory method per il servizio counter. */
    public Counter getCounter() {
        Counter counter = null;
        String counterServiceName = "rmi:/asw830/Counter";
        String registryHost = "192.168.50.101";
        try {
            /* cerca un riferimento al servizio remoto */
            Registry registry = LocateRegistry.getRegistry(registryHost);
            counter = (Counter) registry.lookup(counterServiceName);
        } catch (Exception e) { ... gestisci e ... }
        return counter;
    }
}
```


Esempio: Counter – client (client)

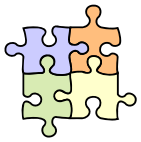
```
package asw.asw830.counter.client;

import asw.asw830.counter.*;

import java.rmi.RemoteException;

public class CounterClient {

    private Counter counter;  /* il servizio */

    public CounterClient(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        /* usa il servizio remoto */
        try {
            for (int i=0; i<25; i++) {
                int result = counter.getCounter();
                ... il contatore ora vale result ...
                ... introduci un ritardo (per convenienza) ...
        }
        } catch (RemoteException e) { ... gestisci e ... }
    }
}
```

Esempio: Counter – client (client)

```
package asw.asw830.counter.client;
import asw.asw
import java
public
    p
    pu
    }
    public void run() {
        /* usa il servizio remoto */
        try {
            for (int i=0; i<25; i++) {
                int result = counter.getCounter();
                ... il contatore ora vale result ...
                ... introduci un ritardo (per convenienza) ...
        }
        } catch (RemoteException e) { ... gestisci e ... }
    }
}
```

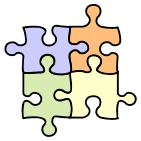
Ciascuna esecuzione concorrente del client ottiene numeri diversi – in una sequenza sempre crescente.

Sia se il client viene eseguito da uno stesso host, sia se viene eseguito da host diversi.



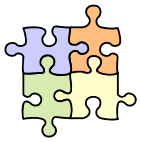
- Servizi stateful

- Un *servizio stateful* è un servizio che gestisce lo *stato conversazionale*, ovvero lo stato delle sessioni – ogni *sessione* è relativa ad uno specifico client, ed ha uno stato indipendente da quello delle altre sessioni
  - una possibile implementazione
    - un oggetto server remoto condiviso gestisce un oggetto "sessione" per ciascuno dei suoi client
    - ciascuna richiesta comprende (ovvero, deve comprendere) un "id di sessione" (come parametro)
    - il server remoto usa una mappa per capire quale oggetto sessione va usato nell'ambito di una richiesta da parte di uno dei suoi client



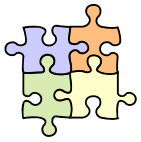
Servizi stateful

- Un'altra possibile implementazione di un servizio stateful – attenzione, esistono anche altre modalità di implementazione
  - viene usato un oggetto remoto condiviso – registrato sull'object registry – che non è il vero servizio ma piuttosto è una factory
    - questa *factory* fornisce solo un'operazione per creare, ogni volta, un nuovo oggetto servente remoto dedicato
    - ciascun nuovo servente remoto sarà relativo a un solo client – ovvero questi serventi remoti non saranno condivisi
  - ciascun client usa l'object registry per accedere alla factory
    - per prima cosa, chiede alla factory di creare per lui un oggetto servente remoto dedicato alla sua sessione
    - successivamente, comunicherà solo con il servente remoto a lui dedicato
  - poiché ciascun servente remoto è legato strettamente al suo client, può mantenere informazioni sullo stato della sessione



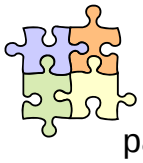
Esempio: SessionCounter – interfaccia

```
package asw.asw830.sessioncounter;

import java.rmi.*;

/* Interfaccia del contatore di sessione. */
public interface SessionCounter extends Remote {
    /* Incrementa e restituisce il valore del contatore di sessione. */
    public int getCounter() throws RemoteException;
}
```



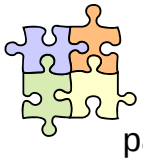
Esempio: SessionCounter – interf. factory

```
package asw.asw830.sessioncounter;

import java.rmi.*;

/* Interfaccia della factory del contatore di sessione. */
public interface SessionCounterFactory extends Remote {
    /* Restituisce un nuovo contatore di sessione. */
    public SessionCounter getSessionCounter()
        throws RemoteException;
}
```

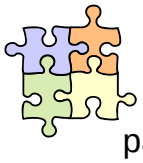
Esempio: SessionCounter – servente (v1)

```
package asw.asw830.sessioncounter.impl;

import asw.asw830.sessioncounter.*;

import java.rmi.*;
import java.rmi.server.*;

/* Implementazione del contatore di sessione. */
public class SessionCounterImpl implements SessionCounter {
    private int counter;    // il contatore di sessione

    /* Crea un nuovo contatore di sessione, inizialmente nullo. */
    public SessionCounterImpl() throws RemoteException {
        super();
        this.counter = 0;
        /* esporta l'oggetto come oggetto RMI */
        UnicastRemoteObject.exportObject(this, 0);
    }

    /* Incrementa e restituisce il valore del contatore di sessione. */
    public int getCounter() {    counter++;    return counter;    }
}
```



Esempio: SessionCounter – servente (v2)

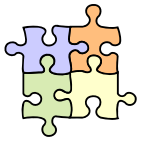
```
package asw.asw830.sessioncounter.impl;

import asw.asw830.sessioncounter.*;

import java.rmi.*;
import java.rmi.server.*;

/* Implementazione del contatore di sessione (versione alternativa). */
public class SessionCounterImpl extends UnicastRemoteObject
                        implements SessionCounter {

    private int counter;    // il contatore di sessione

    /* Crea un nuovo contatore di sessione, inizialmente nullo. */
    public SessionCounterImpl() throws RemoteException {
        super();
        this.counter = 0;
    }

    /* Incrementa e restituisce il valore del contatore di sessione. */
    public int getCounter() {    counter++;    return counter;    }
}
```

Esempio: SessionCounter – servente

```
package asw.asw830.sessioncounter.impl;

import asw.asw830.sessioncou

import jav
impor

/* In
publi

    private int counter;    // il contat

    /* Crea un nuovo contatore di sessione, inizialmente nullo. */
    public SessionCou    Impl() throws RemoteException {
        super();
        this.count    = 0;
    }

    /* Incrementa e restituisce il valore del contatore di sessione. */
    public int getCounter() {    counter++;    return counter;    }
}
```

Il metodo non deve essere dichiarato synchronized.
Infatti, ciascuna istanza di questa classe sarà utilizzata da un solo client.



Esempio: SessionCounter – impl. factory

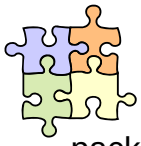
```
package asw.asw830.sessioncounter.impl;

import asw.asw830.sessioncounter.*;

import java.rmi.*;

public class SessionCounterFactoryImpl implements SessionCounterFactory {

    public SessionCounterFactoryImpl() {
        super();
    }

    /* Restituisce un nuovo contatore di sessione. */
    public synchronized SessionCounter getSessionCounter()
            throws RemoteException {
        /* ogni volta crea un nuovo contatore di sessione */
        return new SessionCounterImpl();
        /* essendo un tipo remoto, restituisce un riferimento remoto */
    }

}
```


Esempio: SessionCounter – server (1)

```
package asw.asw830.sessioncounter.server;

import asw.asw830.sessioncounter.*;
import asw.asw830.sessioncounter.impl.*;

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

public class SessionCounterServer {

   public static void main(String[] args) {
      ... crea il servente (la factory) ...
      ... registra il servente (la factory) ...
   }

}
```

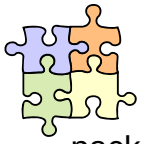
Esempio: SessionCounter – server (2)

```
public static void main(String[] args) {
    /* crea il servente (la factory) */
    SessionCounterFactory sessionCounterFactory =
        new SessionCounterFactoryImpl();
    ... crea il security manager ...
    /* registra il servente (la factory) */
    try {
        SessionCounterFactory stub =
            (SessionCounterFactory)
                UnicastRemoteObject.exportObject(sessionCounterFactory, 0);
        String sessionCounterFactoryName =
            "rmi:/asw830/SessionCounterFactory";
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(sessionCounterFactoryName, stub);
        /* servente (factory) registrato */
    } catch (Exception e) {
        ... gestisci eccezione e ...
    }
}
```

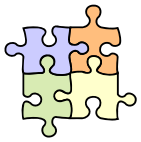
Esempio: SessionCounter – client (main)

```
package asw.asw830.sessioncounter.client;

import asw.asw830.sessioncounter.*;
import asw.asw830.counter.client.connector.ServiceFactory;

/* Applicazione client: ottiene e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto SessionCounterClient. */
    public static void main(String[] args) {
        SessionCounter sessionCounter =
            ServiceFactory.getInstance().getSessionCounter();
        SessionCounterClient client = new SessionCounterClient(sessionCounter);
        client.run();
    }

}
```



Esempio: SessionCounter – client (service f.)

```
package asw.asw830.sessioncounter.client.connector;

import asw.asw830.counter.*;

import java.rmi.*;

/* Factory per il servizio SessionCounter. */
public class ServiceFactory {

    ... variabile, costruttore e metodo per singleton ...

    /* Factory method per la factory del servizio session counter. */
    public SessionCounterFactory getSessionCounterFactory() {
        SessionCounterFactory factory = null;
        String factoryName = "rmi:/asw830/SessionCounterFactory";
        String registryHost = "192.168.50.101";
        try {
            /* cerca un riferimento al servizio remoto */
            Registry registry = LocateRegistry.getRegistry(registryHost);
            factory = (SessionCounterFactory) registry.lookup(factoryName);
        } catch (Exception e) { ... gestisci e ... }
        return factory;
    }
    ... segue ...
```

Esempio: SessionCounter – client (service f.)

```
    /* Factory method per il servizio session counter. */
    public SessionCounter getSessionCounter() {
        SessionCounter sessionCounter = null;
        try {
            /* prima ottiene la factory al servizio remoto */
            SessionCounterFactory factory = getSessionCounterFactory();
            /* poi ottiene il session counter */
            sessionCounter = factory.getSessionCounter();
        } catch (Exception e) { ... gestisci e ... }
        return sessionCounter;
    }
}
```

```
/* Restituisce un nuovo contatore di sessione. */
public synchronized SessionCounter getSessionCounter() {
    return new SessionCounterImpl();
}
```

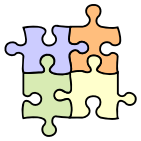
Esempio: SessionCounter – client (client)

```
package asw.asw830.sessioncounter.client;

import asw.asw830.sessioncounter.*;

import java.rmi.RemoteException;

public class SessionCounterClient {

    private SessionCounter counter;  /* il servizio */

    public SessionCounterClient(SessionCounter counter) {
        this.counter = counter;
    }

    public void run() {
        /* usa il servizio remoto */
        try {
            for (int i=0; i<25; i++) {
                int result = counter.getCounter();
                ... il contatore ora vale result ...
                ... introduci un ritardo (per convenienza) ...
            }
        } catch (RemoteException e) { ... gestisci e ... }
    }
}
```


Esempio: SessionCounter – client (client)

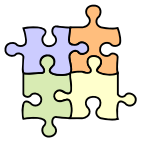
```
package asw.asw830.sessioncounter.client;

import asw.asw830.sessioncounter.*;

import java.rmi.RemoteException;

public class SessionCounterClient {

    private SessionCounter counter;  /* il servizio */

    public SessionCounterClient(SessionCounter counter) {
        this.counter = counter;
    }

    public void run() {
        /* usa il servizio remoto */
        try {
            for (int i=0; i<25; i++) {
                int result = counter.getCounter();
                ... il contatore ora vale result ...
                ... introduci un ritardo (per convenienza) ...
            }
        } catch (RemoteException e) { ... gestisci e ... }
    }
}
```

Ora ciascuna esecuzione concorrente del client ottiene la stessa sequenza di numeri, a partire da 1.



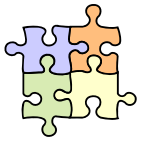
Esempio: SessionCounter – discussione

- Nell'esempio mostrato
  - Main (lato client) ottiene dalla service factory direttamente un session counter – e lo inietta nel client
  - la service factory (lato client) ricerca prima la factory remota e poi il session counter
  - il client usa direttamente il session counter
- In alternativa
  - Main ottiene dalla service factory la factory remota per il session counter – e la inietta nel client
  - il client deve prima creare un session counter tramite la factory remota – e poi può usarlo
- Alcuni strumenti di middleware operano in quest'ultimo modo
  - l'accesso a un servizio remoto richiede talvolta due operazioni distinte, per ottenere prima la factory e poi il servizio



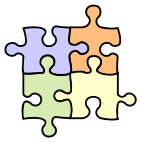
- Gestione dello stato

- Quali sono le possibilità per la gestione dello stato dell'applicazione?
- Quali sono le possibilità per la gestione dello stato della sessione?
- Quali sono le possibilità se bisogna gestire sia stato dell'applicazione che stato della sessione?
- Quali i possibili ruoli nella gestione dello stato per
  - l'oggetto client?
  - l'oggetto servente?
  - eventuali altri oggetti remoti?
  - eventuali altri oggetti remoti condivisi?
  - l'eventuale base di dati?



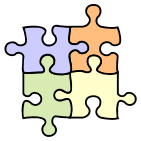
- Legame dei parametri

- RMI è di solito basato su un meccanismo di legame dei parametri per valore-risultato
  - in cui, in un metodo, si fa distinzione tra parametri di input (*in*) e risultati/parametri di output (*out*)
- Esempio: calcola le radici di un'equazione di secondo grado
  - intestazione del metodo (omettendo i tipi)
    - *calcolaRadici(in* a, *in* b, *in* c, *out* x1, *out* x2)
  - invocazione del metodo
    - calcolaRadici(1, 4, 2, radice1, radice2)
- In Java RMI – che è basato su Java – questo tipo di legame è disponibile solo in una versione limitata – infatti è possibile un solo risultato, "legato" mediante un'assegnazione
  - double sqrt(double x)
  - r = sqrt(125)



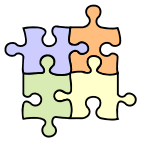
Legame dei parametri per valore-risultato

- *Legame dei parametri per valore-risultato*
  - al momento dell'invocazione, vengono legati i parametri di input
    - i parametri attuali di input (lato client) sono valori – questi valori sono usati come valori iniziali per i parametri formali (lato server), che sono variabili
  - quando termina l'esecuzione del metodo, vengono legati i risultati – ovvero, i parametri di output
    - i risultati di output (lato server) sono valori – questi valori sono usati come valori da assegnare ai parametri attuali di output (lato client), che sono (devono essere) variabili



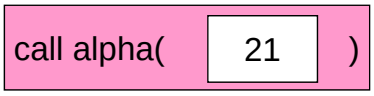
Legame dei parametri per valore-risultato

- In Java RMI, il legame dei parametri e dei risultati avviene con modalità diverse, a secondo del tipo del parametro o risultato
  - tipi primitivi – sono legati per valore
  - tipo riferimento "remoto" – ovvero, estende *java.rmi.Remote*
    - i riferimenti remoti sono riferimenti univoci in rete
    - il riferimento remoto viene legato per valore – ovvero, viene passato il riferimento remoto all'oggetto remoto
    - come effetto collaterale dell'esecuzione di operazioni remote, è possibile modificare lo stato dell'oggetto remoto
  - tipo riferimento "locale" – ovvero, non remoto
    - i riferimenti locali hanno senso solo localmente al processo in cui sono definiti – non vengono passati per valore
    - l'oggetto viene *serializzato* (insieme a tutti gli oggetti serializzabili da esso raggiungibili) e legato per valore
    - attenzione, niente effetti collaterali!

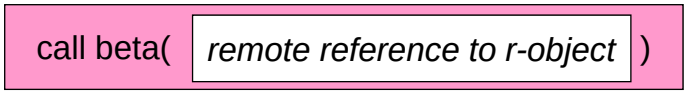


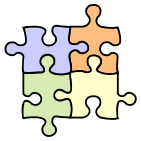
Legame dei parametri per valore-risultato

- Un parametro di un tipo primitivo viene legato per valore
  - alpha(21)



- Un parametro di un tipo riferimento "remoto" viene legato per valore – il valore è il riferimento remoto
  - beta(r-object)





Legame dei parametri per valore-risultato

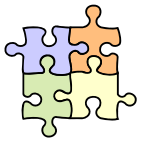
- Un parametro di un tipo riferimento "locale" viene legato per valore – il valore è una copia serializzata dell'oggetto – che viene trasmessa e poi "ricostruita" dal servente
  - gamma(l-object)

call gamma(

```
{
   "type" : "asw.model.Employee",
   "id" : "410",
   "firstName" : "John",
   "lastName" : "Smith",
   "dept" :
   {
      "type" : "asw.model.Department",
      "id" : "992",
      "name" : "DIA"
   }
}
```

)

- *Serialization*
  - is the process of translating a data structure or object state into a format that can be stored (e.g., in a file) or transmitted across a network connection link and "resurrected" later in the same or another computer environment (Wikipedia, ottobre 2013)



Legame dei parametri – esercizio 1

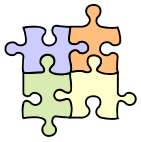
- Si consideri il seguente metodo di un oggetto *o*

```
public void aumentaStipendio(Impiegato i) {
        i.setStipendio( i.getStipendio()*1.1 );
}
```

  - e una sua chiamata

```
o.aumentaStipendio(imp);
```

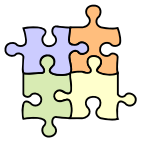
- Che succede nelle seguenti situazioni?
  1. *imp* e *o* sono di tipi locali – le invocazioni dei metodi sono locali
  2. *imp* è di un tipo locale, ed è locale al client, ma *o* è di un tipo remoto
  3. *imp* e *o* sono di tipi remoti, anche le chiamate dei metodi get e set sono remote



Legame dei parametri – esercizio 1

```
o.aumentaStipendio(imp);
```

- Che succede nelle seguenti situazioni?
  1. *imp* e *o* sono di tipi locali – le invocazioni sono locali
     - lo stipendio dell'oggetto *imp* originale locale viene incrementato
  2. *imp* è di un tipo locale, è locale al client, ma *o* è remoto
     - viene incrementato lo stipendio del clone remoto serializzato di *imp*
     - lo stipendio dell'oggetto *imp* originale locale rimane invariato
  3. *imp* e *o* sono di tipi remoti, anche le chiamate dei metodi get e set sono remote
     - lo stipendio dell'oggetto *imp* originale remoto viene incrementato
     - attenzione: tre chiamate remote!



Legame dei parametri – esercizio 2

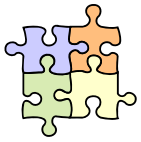
- Si consideri il seguente metodo di un oggetto *o*

```
public Impiegato trovaImpiegato(Matricola m) {
        ...
        return i;
}
```

  - e una sua chiamata

```
Impiegato imp = o.trovaImpiegato(m);
imp.setStipendio(0);
```

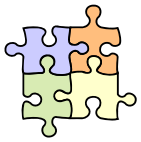
- Che succede nelle seguenti situazioni?
  1. *imp* e *o* sono di tipi locali – le invocazioni sono locali
  2. *o* è remoto, ma il tipo *Impiegato* è locale (non è remoto)
  3. *imp* e *o* sono di tipi remoti, anche le chiamate dei metodi get e set sono remote



Legame dei parametri – esercizio 2

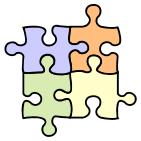
```
Impiegato imp = o.trovaImpiegato(m);
imp.setStipendio(0);
```

- Che succede nelle seguenti situazioni?
  1. *imp* e *o* sono di tipi locali – le invocazioni sono locali
     - lo stipendio dell'oggetto *imp* originale locale viene azzerato
  2. *o* è remoto, ma il tipo *Impiegato* è locale (non è remoto)
     - viene azzerato lo stipendio del clone locale serializzato di *imp*
     - lo stipendio dell'oggetto *imp* originale remoto rimane invariato
  3. *imp* e *o* sono di tipi remoti, anche le chiamate dei metodi get e set sono remote
     - lo stipendio dell'oggetto *imp* originale remoto viene azzerato
     - due chiamate remote



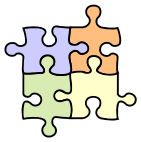
Discussione

- Dagli esempi mostrati, risulta che la semantica di una chiamata remota
  - coincide con quella di una chiamata locale, se tutti i tipi riferimento sono remoti
  - può essere diversa da quella di una chiamata locale, se invece sono coinvolti tipi riferimento locali
- Alcune ulteriori considerazioni su questo aspetto
  - non è comune che tutti i tipi riferimento siano remoti – poiché il costo temporale di una chiamata remota è molto maggiore di quello di una chiamata locale
  - attenzione alla "moltiplicazione" delle chiamate remote – ed al costo temporale aggiuntivo di serializzazioni e deserializzazioni
  - in ogni caso, se non sono previsti effetti collaterali, la semantica di una chiamata remota coincide con quella di una chiamata locale anche se sono coinvolti tipi riferimento locali



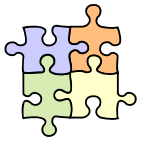
Value Object e Value Object Assembler

- Due design pattern [Core J2EE Patterns] per applicazioni ad oggetti distribuiti – noti anche con nomi diversi
  - *Value Object – Data Transfer Object*
    - un oggetto che trasporta i dati di uno o più Domain Object tra processi
  - *Value Object Assembler*
    - un oggetto che sa trasformare i dati di uno o più Domain Object in un Value Object
    - può anche offrire le funzionalità (complementari) di creare DO da un VO oppure aggiornare dei DO a partire da un VO
- Sono pattern utili per rappresentare in un processo informazioni di oggetti che vivono in altri processi
  - usandoli, si è coscienti che si stanno gestendo delle copie
  - i VOA sostengono la gestione delle copie



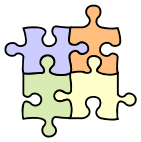
- Esercizio

- Scrivere un'applicazione client-server RMI, in cui
  - il client comunica al server, uno alla volta, gli elementi numerici di una sequenza, per farglieli sommare – e alla fine chiede al server la somma calcolata
  - deve poter essere eseguita da più client, tra loro indipendenti, in modo concorrente
- Osservazione – per descrivere l'interfaccia del server
  - non è sufficiente descrivere il prototipo di ciascun metodo remoto
  - è necessario anche definire un opportuno protocollo (contrattuale) per un uso corretto del servizio



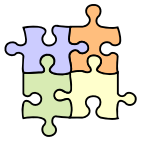
- Garbage collection e gestione della memoria

- Per gli oggetti distribuiti, viene usato un meccanismo di garbage collection distribuito
  - ogni oggetto distribuito sa quanti oggetti ne conoscono il riferimento
  - può essere deallocato se non è referenziato da nessuno
- Possibili anche meccanismi di attivazione/passivazione di oggetti



- Callback

- Abbiamo visto come un oggetto client può richiedere servizi a un oggetto servente
  - talvolta è utile poter consentire ad un servente di contattare i propri client
    - ad es., per realizzare una comunicazione publisher/subscriber – il servente notifica degli eventi ai propri client – l'alternativa è il polling effettuato dai client
    - ad es., per consentire al servente di accedere allo stato della sessione se questo è memorizzato presso il client – con un meccanismo, ad es., analogo a quello dei cookie
- La soluzione è offerta dalla possibilità di realizzare un meccanismo di *callback*
  - in cui il servente "richiama" il client
  - consente la comunicazione bidirezionale



Callback

- Il callback può essere implementato come segue
  - il client crea un oggetto remoto (*oggetto callback*) che implementa un'interfaccia remota che il servente può richiamare
  - il servente fornisce un'operazione che consente al client di specificare il proprio oggetto callback
  - quando il servente ne ha bisogno, può contattare l'oggetto callback