UNIVERSITÀ DI PISA

Programmazione di reti
Corso B

22 Novembre 2016
Lezione 9

Eccezione ECHO server NIO

```
if (key.isReadable()){
  try{
     SocketChannel channel= (SocketChannel) key.channel();
     ByteBuffer[] bfs = (ByteBuffer[]) key.attachment();
     long response=channel.read(bfs);
     if (response==-1){
        channel.close();
        key.cancel();
        continue; //!!!!!!!!!!!!
     }
     if (!bfs[0].hasRemaining()){
        bfs[0].flip();
        int l=bfs[0].getInt();
        if(bfs[1].position()==l){
           bfs[1].flip();
           channel.register(selector, SelectionKey.OP_WRITE,bfs[1]);
        }
     }
  } catch(IOException e){
     e.printStackTrace();
  }

}
if (key.isWritable()){//!!!!!!!!!!!!!!!!
```

Contenuti

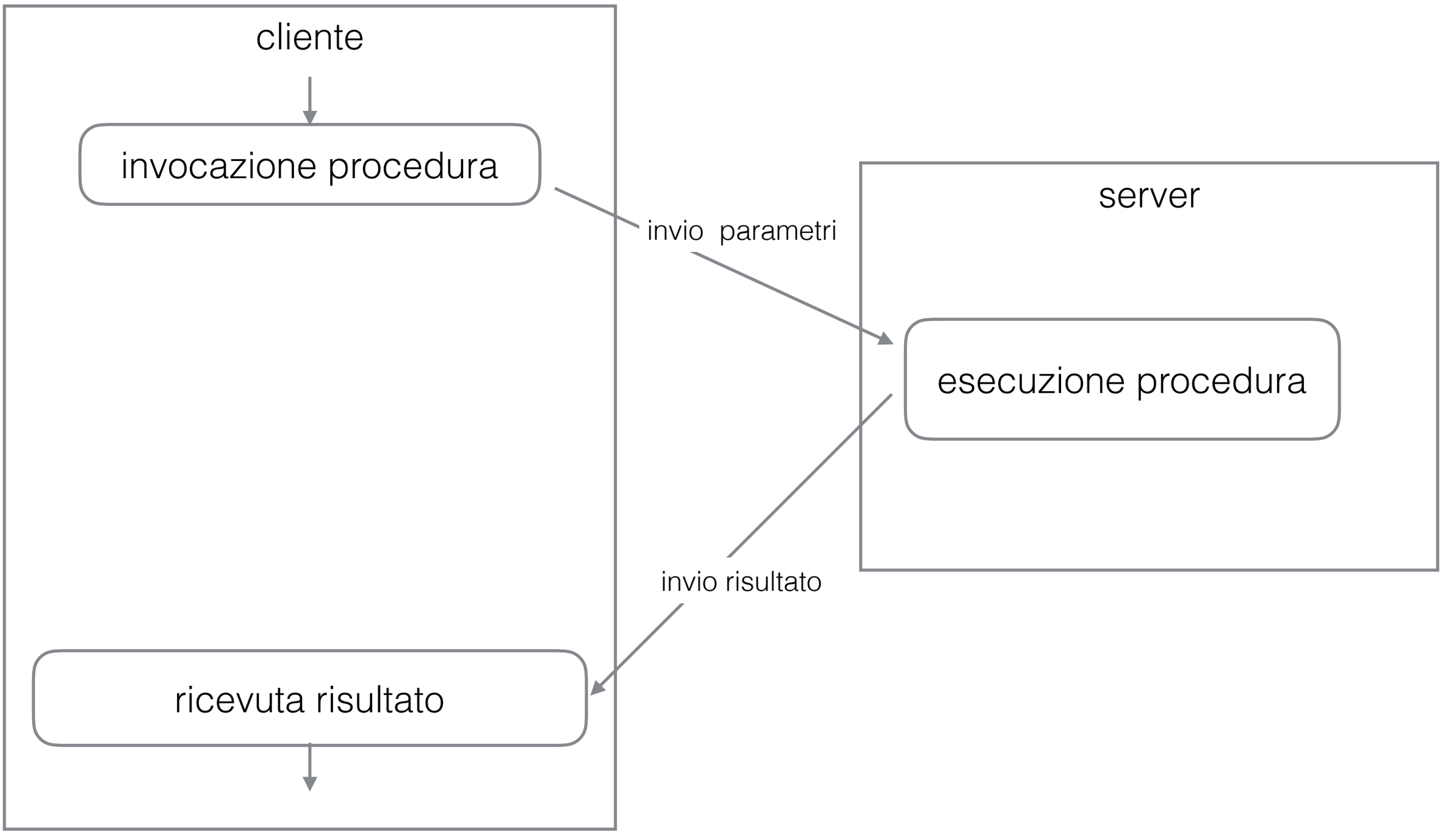
- Introduzione a RMI *(remote method invocation)*
  - Concetti
  - Componenti
  - Esempi

Programmazione di rete ad alto livello

- *Socket* - programmazione di rete a basso livello
  - programmatore deve implementare sia la logica dell'applicazione, che la comunicazione tramite la rete (protocollo, invio messaggi, validità etc)

- Una paradigma alternativa - *Remote Procedure Call* (RPC)
  - interfaccia a livello di procedura: il *server* mette a disposizione un insieme di procedure (con varie funzionalità). Il cliente può invocare le procedure inviando i parametri richiesti.
  - programmatore implementa le procedure e le invocazioni dal cliente, senza dover occuparsi della parte di rete (invio messaggi, protocollo, etc)

RPC


cliente
invocazione procedura
invio parametri
server
esecuzione procedura
invio risultato
ricevuta risultato

RPC

- Semplifica la programmazione di rete: programmatore non si deve occupare dell'invio dei messaggi

- Però:

    - Parametri e risultati possono essere solo dati primitivi

    - Programmazione procedurale - non può usare vantaggi di OOP

    - Cliente deve conoscere l'indirizzo del *server* dove esegue il codice remoto

- **Java RMI** risolve questi problemi - introduce gli oggetti, usa serializzazione Java per poter trasferire qualsiasi tipo di dati, usa *dynamic class loading* per offrire comportamento polimorfico. Il cliente deve solo conoscere l'indirizzo di un *registry* (registro) e il nome con cui può cercare oggetti distribuiti.

Java *Remote Method Invocation*

- Comunicazione tra JVM: un oggetto sulla JVM cliente può invocare metodi di un oggetto sulla JVM *server*

- Paradigma RMI è simile a RPC pero usando metodi e oggetti, invece di procedure, con serializzazione Java.

- *Runtime* RMI - include il sistema che implementa la comunicazione tra cliente e *server* attraverso la rete

- Il programmatore - non deve preoccuparsi dei dettagli relativi ai *socket*, messaggi, connessioni, etc.: sembra di lavorare con oggetti Java normali.

JVM cliente

riferimento
oggetto remoto

invocazione metodo

serializzazione parametri
(*marshalling*)

invio invocazione e parametri

deserializzazione risultato
(*unmarshalling*)

invio risultato

JVM *server*

oggetto remoto

deserializzazione parametri
(*unmarshalling*)

esecuzione metodo

serializzazione risultato
(*marshalling*)

Struttura applicazione RMI

- *Server* crea e esporta **oggetti remoti**
- *Server* pubblica i riferimenti a questi oggetti nel *RMI Registry* - (nome: riferimento)
- Cliente cerca nel *RMI Registry* un oggetto remoto usando il nome e ottiene il riferimento
- Cliente invoca metodi dell'oggetto remoto usando il riferimento

RMI Registry

(nome1, riferimento1)
(nome2, riferimento2)
…..

Cliente

nome

riferimento oggetto remoto

invocazione metodo

utilizzo risultati

ricerca nome

invio risultato della ricerca: il riferimento

registrazione (nome, riferimento)

invio invocazione e parametri

invio risultato

Server

Creazione oggetto remoto

Creazione e esportazione riferimento all'oggetto remoto

Invocazione metodo

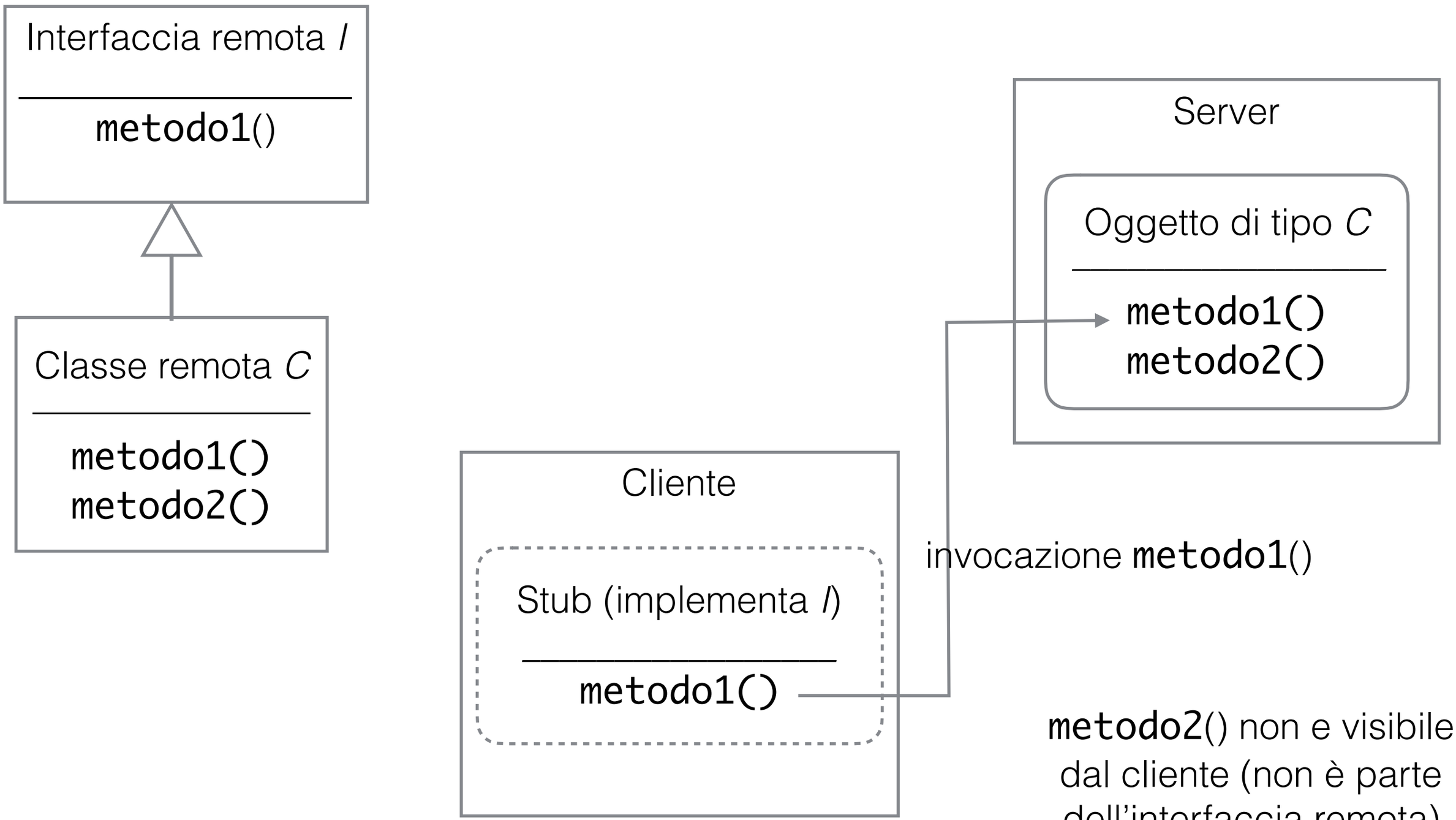
Oggetti remoti (distribuiti)

- Concetto alla base di RMI
- Oggetti di una JVM cui metodi possono essere invocati da un'altra JVM (anche a distanza)
- RMI offre il framework per lavorare con questi oggetti:
  - pubblicazione e accesso a oggetti distribuiti tramite il *RMI registry*
  - scambiare informazioni sulle classi degli oggetti e la loro definizione, nonché caricare le definizioni di una classe per istanziarla, usando *dynamic class loading* e *code mobility*
  - scambiare dati degli oggetti tramite la rete usando serializzazione
  - tutto in modo trasparente per il programmatore

Oggetti remoti

- Implementano una *interfaccia remota* (un set di metodi accessibili a distanza).
- Esistono solo su una parte dei nodi della rete
- Possono essere usati in qualsiasi nodo
    - sui nodi dove l'oggetto non esiste, viene inviata una *stub* - funziona come *proxy* all'oggetto - vista attraverso l'interfaccia remota.

Oggetti remoti



Interfaccia remota

- Ogni interfaccia che:
  - Implementa interfaccia `Remote`
  - per tutti i metodi dichiara di lanciare `RemoteException`

Classe remota

- Implementa una o più interfacce remote: implementa tutti i metodi delle interfacce
- Ogni istanza è un'oggetto remoto
- Deve sovrascrivere i metodi `equals()`, `hashCode()` e `toString()`
  - la verifica dell'uguaglianza tra due oggetti remoti dovrebbe accedere all'oggetto stesso tramite la rete, però il metodo `equals()` non lancia `RemoteException`, quindi l'invocazione deve essere locale. Due oggetti remoti sono considerati uguali se i loro riferimenti sono uguali.
  - il *hashCode* generato è lo stesso per riferimenti allo stesso remote object
  - la stringa generata da `toString()` contiene informazioni sul oggetto remoto
  - la sovrascrittura si può fare semplicemente estendendo la classe `RemoteObject` che re-implementa questi metodi per RMI

Lo *stub* (*proxy*)

- Un riferimento ad un oggetto remoto
- Istanza di una classe che implementa tutti i metodi dell'interfaccia remota
- Il codice dello *stub* viene generato automaticamente quando si fa l'esportazione dell'oggetto remoto.
- Il codice dei metodi dello *stub* non è uguale al codice della classe remota. Implementa il processo di invocazione del metodo reale: inviare parametri sulla rete e ricevere risultato usando *socket* e serializzazione.
- Lo *stub* contiene informazioni sul *server* e l'oggetto esportato (IP, port, objectID)

Trasferimento oggetti

- I parametri dei metodi o i risultati da restituire devono essere trasferiti tra cliente e *server*

- RMI si occupa del trasferimento, usando serializzazione Java.

- Ogni oggetto che deve essere trasferito deve essere un dato primitivo, implementare l'interfaccia `Serializable` o essere un oggetto remoto.

- Oggetti remoti vengono passati come riferimento (lo *stub*)

- Altri oggetti vengono passati come valore (una copia)

Sviluppare un *server* RMI

- Definire un'interfaccia remota `I`
- Definire una classe remota `C` che implementa `I`
- Creare oggetti di tipo `C`
- Esportare gli oggetti
- Registrare gli oggetti in un *RMI Registry*
- Esempio: gestione studenti - ricerca usando il nome, aggiornamento del voto

Interfaccia remota

Estende interfaccia **Remote**

```
public interface StudentManager extends Remote {

    public final static String REMOTE_OBJECT_NAME="PISA_STUDENTS";

    public ArrayList<Student> searchByLastName(String lastName)
            throws RemoteException;

    public boolean setStudentGrade(int studentId, double grade)
            throws RemoteException;
    }
```

Tutti i metodi lanciano **RemoteException**

Classe Student - serializzabile

```
public class Student implements Serializable{

   private static final long serialVersionUID = 2L;

   private String fname, lname;
   private String streetAddress;
   private int addressNo;
   private double grade;
   private int studentId;
   public Lock lock;

   public Student(String fname, String lname,String street, int no,int id){
      this.fname=fname;
      this.lname=lname;
      this.streetAddress= street;
      this.addressNo=no;
      this.grade=-1;
      this.studentId=id;
      this.lock= new ReentrantLock();
}
```

```
public String toString(){
   this.lock.lock();
   StringBuilder sb= new StringBuilder();
   sb.append(this.fname);
   sb.append(" ");
   sb.append(this.lname);
   sb.append(" living at ");
   sb.append(this.addressNo);
   sb.append(" ");
   sb.append(this.streetAddress);
   sb.append(" street. Student id ");
   sb.append(this.studentId);
   sb.append(". Current grade ");
   sb.append(this.grade);
   this.lock.unlock();
   return sb.toString();
}
public String getLname() {
   return this.lname;
}
```

```
public void setGrade(double grade){
   this.lock.lock();
   this.grade=grade;
   this.lock.unlock();
}

public int getId(){
   return this.studentId;
}

public Student copy(){
      Student result= new Student(this.fname,this.lname,this.streetAddress,
            this.addressNo,this.studentId);
      this.lock.lock();
      result.setGrade(this.grade);
      this.lock.unlock();
      return result;
   }

}
```

Classe remota

```
public class PisaStudentManager
extends RemoteObject implements StudentManager {

   private static final long serialVersionUID = 1L;
   ArrayList<Student> students;

   public PisaStudentManager() {
      this.students=new ArrayList<>();
   }

   @Override
   public ArrayList<Student> searchByLastName(String lastName)
         throws RemoteException {
      ArrayList<Student> result= new ArrayList<>();
      for (Student s : this.students){
         if (s.getLname().equals(lastName)){
            result.add(s.copy());
         }
      }
      return result;
   }
```

Implementa interfaccia remota
Estende **RemoteObject**

```
@Override
public boolean setStudentGrade(int studentId, double grade)
      throws RemoteException {
   for (Student s : this.students){
      if (s.getId()==studentId){
         s.setGrade(grade);
         return true;
      }
   }
   return false;
   }

   public void addStudent(Student s){
      this.students.add(s);
   }
}
```

Esportazione

- Usando la classe `UnicastRemoteObject`

`Remote exportObject(Object o, int port)`

esporta l'oggetto `o` al runtime RMI, crea una classe *stub* per l'oggetto e ne restituisce un'istanza. Il sistema RMI crea un *server* che aspetta invocazioni dei metodi dell'oggetto `o`, attivo su *port* `port`. Lo *stub* memorizza il *port* e l'indirizzo di questo *server*. Se `port==0`, il *port* viene assegnato in modo aleatorio.

`RemoteStub exportObject(Object o)`

deprecated da Java 8. Usa classi *stub* statiche : generate usando *rmic* manualmente

`boolean unexportObject(Remote obj, boolean force)`

Rimuove l'oggetto dal sistema RMI.

Registrazione

- Registrazione dello *stub* in un *RMI Registry*
  - Un RMI *registry* è un oggetto remoto disponibile nella libreria RMI - implementa interfaccia remota **`Registry`** con i metodi:

`void bind(String name, Object stub)`

Associa il nome allo *stub*. Solo permesso da *localhost*. Lancia eccezione se una registrazione con lo stesso nome esiste già.

`void rebind(String name, Object stub)`

Associa il nome allo *stub*. Se l'associazione esiste già, viene rimpiazzata. Di solito si usa direttamente questo metodo.

Registrazione

- Interfaccia remota `Registry` (cont):

`void unbind(String name)`

Rimuove un'associazione

`String[] list()`

Restituisce una lista di nomi di oggetti già registrati.

`Remote lookup(String name)`

Restituisce lo *stub* associato con **`name`** nel *registry*

Registrazione

- Classe `LocateRegistry` - metodi statici per avviare un `Registry` o per ottenere un riferimento al *registry* esistente sul *server* (da avviare separatamente)

`Registry createRegistry(int port)`

Crea un oggetto remoto `Registry` disponibile sul port `port`, e restituisce il riferimento (lo *stub*) a questo oggetto.

RMI *Registry*

- Può essere avviato esternamente al programma usando comandi sotto. Per avviarlo sul port 1099:
  - Unix: rmiregistry
  - Windows: start rmiregistry
- Variante con port specificato:
  - Unix: rmiregistry 2000
  - Windows: start rmiregistry 2000
- Da avviare nel *folder bin* (dove sono le definizioni delle interfacce - file .class), o aggiungere il *folder bin* alla *classpath*

Registrazione RMI Registry

- Classe `LocateRegistry` (cont):

`Registry getRegistry()`

Restituisce un riferimento (*stub*) al `Registry` che è stato aperto sul *port* 1099 del *localhost*.

`Registry getRegistry(int port)`

Restituisce un riferimento (*stub*) al `Registry` avviato sul *port* `port`.

`Registry getRegistry(String host)`

Restituisce un riferimento (*stub*) al `Registry` avviato sul *port* 1099 sulla *host* `host`.

`Registry getRegistry(String host, int port)`

Restituisce un riferimento (*stub*) al `Registry` avviato sul *port* `port` sulla *host* `host`.

Questi metodi costruiscono lo *stub* **localmente**, usando i parametri forniti, e non provano di contattare l'oggetto remoto `Registry`. Quindi i metodi non falliscono se non c'è nessun `Registry` all'indirizzo specificato. Un'errore si presenterà solo durante le invocazioni successive.

```
public class StudentManagerMain {

   public static int PORT=2000;

   public static void main(String[] args) {
      //create manager and add data
      PisaStudentManager manager= new PisaStudentManager();
      manager.addStudent(new Student("Robert","Brown","Dawson",12,0));
      manager.addStudent(new Student("Michael","Reds","Pearse",40,1));
      manager.addStudent(new Student("Joanna","Moore","Collins",62,2));
      manager.addStudent(new Student("Ann","Brown","Buffallo",132,3));

      try {
         //export object
         StudentManager managerStub=
               (StudentManager) UnicastRemoteObject.exportObject(manager, 0);

         //register to RMI registry
         Registry registry= LocateRegistry.createRegistry(PORT);
         registry.rebind(StudentManager.REMOTE_OBJECT_NAME, managerStub);

         System.out.println("Finished server setup.");
      } catch (RemoteException e) {
         System.out.println("Error setting up RMI server: "+e.getMessage());
      }
   }
}
```

Sviluppare un cliente RMI

- Scrivere un programma che usa oggetti di tipo `I` (interfaccia)
- Programma deve caricare lo *stub* usando il RMI *Registry*
    - cliente deve conoscere *host* e *port* del RMI *Registry* e il nome dell'oggetto remoto

```
public class StudentClient {

   public static int REGISTRY_PORT=2000;
   public static String REGISTRY_HOST="localhost";

   public static void main(String[] args) {
      try {
         Registry registry =
            LocateRegistry.getRegistry(REGISTRY_HOST, REGISTRY_PORT);
         StudentManager manager=
            (StudentManager) registry.lookup(StudentManager.REMOTE_OBJECT_NAME);
         System.out.println(manager);
```

Ottenere stub dal RMI Registry

Usare stub per invocare metodi dell'oggetto remoto

```
ArrayList<Student> students= manager.searchByLastName("Brown");
System.out.println("Received students:");
for(Student s: students){
   System.out.println(s);
}

manager.setStudentGrade(0, 30);
students= manager.searchByLastName("Brown");
System.out.println("Received students:");
      for(Student s: students){
         System.out.println(s);
      }

      } catch (RemoteException e) {
         System.out.println("Error in client: "+e.getMessage());
      } catch (NotBoundException e) {
         System.out.println("Remote object not found: "+e.getMessage());
      }
   }
}
```

```
Proxy[StudentManager,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[10.200.2.57:52834](remote),objID:[-1cebffc0:154589b5610:-7fff,
-7511745404030969195]]]]]
Received students:
Robert Brown living at 12 Dawson street. Student id 0. Current grade -1.0
Ann Brown living at 132 Buffallo street. Student id 3. Current grade -1.0
Received students:
Robert Brown living at 12 Dawson street. Student id 0. Current grade 30.0
Ann Brown living at 132 Buffallo street. Student id 3. Current grade -1.0
```

cliente

```
LocateRegistry.getRegistry(“localhost”
,2000)
```

↓ crea localmente!

`Registry`
(*stub*, conosce *port* 2000)

```
registry.lookup(“PISA_STUDENTS”)
```

Invio richiesta

server

PISA _STUDENTS:

`StudentManager`
(*stub*, conosce *port* x)

`Registry` (su *port* 2000)

Invio riferimento (*stub*)

`StudentManager`
(*stub*, conosce *port* x)

```
manager.searchByName(“Brown”)
```

Invio richiesta

`PisaStudentManager`
(su *port* anonimo x)

```
for(Student s: students)
```

Invio risultato (`ArrayList`)

Dettagli Java RMI

- La comunicazione avviene in modo sincrono:
  - L'invocazione di un metodo di un'oggetto remoto è bloccante: il *thread* cliente aspetta fin che il risultato dal *server* viene ricevuto.
- Il protocollo usato a livello di trasporto è TCP
  - Affidabilità, *stream* di dati continuo, però più pesante.
  - Per ogni invocazione di un metodo si apre una nuova connessione TCP.

Dettagli Java RMI

- Gli oggetti sono:
  - di tipo *unicast* - solo una copia dell'oggetto esiste (a differenza di multicast, quando si mantengono più copie per migliore performance - reattività, *fault-tolerance*)
  - volatili - vengono cancellati quando il processo che gli ha creati finisce (a differenza degli oggetti persistenti, che possono essere salvati e ripristinati più tardi)
  - non-rilocabili - non possono essere spostati da un JVM ad un altro

Dettagli Java RMI

- *Failure semantics* - cosa succede se metodo fallisce
  - per programmi Java locali: se si invoca un metodo di un oggetto, questo viene eseguito esattamente una volta (semantica *exactly-once*)
    - se si riceve un risultato, il metodo è stato eseguito esattamente una volta
  - per programmi Java RMI, un metodo viene eseguito al massimo una volta (semantica *at-most-once*)
    - problemi se server diventa inattivo o la connessione viene interrotta.
    - se cliente riceve un risultato, il metodo è stato eseguito esattamente una volta
    - se cliente riceve un'eccezione, il metodo è stato eseguito una o zero volte.
- altre semantiche non applicabile a RMI:
  - *maybe*: metodo viene eseguito al massimo una volta, però senza controlli di *fault-tolerance*.
  - *at-least-once*: metodo viene eseguito fin che il cliente riceve la risposta (se la risposta si perde si esegue il codice di nuovo). Metodi devono essere in grado di ripetere la stessa azione con lo stesso risultato.

Callback

- RMI invoca metodi in modo sincrono
- A volte non è molto efficiente aspettare che un'operazione finisca:
    - stampare un documento - il metodo `stampa()` può restituire un risultato positivo se la stampa è iniziata, però il cliente verra notificato più tardi quando la stampa sarà finita.
    - sistemi di *chat*: un utente viene notificato quando un amico invia un messaggio, senza dover essere bloccato in un'invocazione di un metodo
    - etc.
- In RMI, è il cliente che inizia la comunicazione, invocando il metodo del *server*.
- Il meccanismo di *callback*: abilita il server di iniziare una comunicazione, per notificare il cliente di un evento.

Callback

- Quando un cliente invoca un metodo sul *server*, invia dei parametri
- Questi parametri possono essere oggetti “normali” ma anche oggetti remoti (viene inviato uno *stub*)
- Gli oggetti remoti inviati possono essere invocati dal *server* - in questo modo il server può iniziare una comunicazione col cliente

RMI *callback*

- Il cliente esporta un oggetto remoto con uno o più metodi

- Il cliente contatta il *server* e invia questo oggetto (riferimento, *stub*) usando l'oggetto remoto pubblicato dal *server* (come parametro al metodo remoto)

- Il *server* può adesso contattare il cliente quando necessario, usando il metodo definito dal cliente (il cliente decide cosa succede quando viene contattato)

Esempio: *chat room* con RMI

cliente (host h2)

`Registry`
stub, conosce host h1, port 1099

`ChatRoom`
stub, conosce host h1, port p1

`PisaChatClient`
(su port anonimo p2)

server (host h1)

"CHAT_ROOM":

`ChatRoom`
(stub, conosce h1, p1)

`Registry` (su port 1099)

`PisaChatRoom`
(su port anonimo p1)

`ChatClient`
stub, conosce host h2, port p2

Interfaccia per oggetto remoto *server*

```
public interface ChatRoom extends Remote {
   public static final String OBJECT_NAME="CHAT_ROOM";

   public int enter(ChatClient c) throws RemoteException;
   public int exit(ChatClient c) throws RemoteException;
   public int message(String message) throws RemoteException;
}
```

Metodi che permettono ai clienti di entrare e lasciare la *chat room*, e inviare dei messaggi.

Interfaccia per oggetto remoto cliente

```
public interface ChatClient extends Remote{
   public void message(String message) throws RemoteException;
}
```

Metodo che permette al *server* di inviare un messaggio al cliente .

Classe per oggetto remoto *server*

```
public class PisaChatRoom extends RemoteObject
implements ChatRoom{
   private static final long serialVersionUID = 8351397577318521206L;

   private Set<ChatClient> users;

   public PisaChatRoom() {
      this.users= Collections.synchronizedSet(new HashSet<>());
   }

   @Override
   public int enter(ChatClient c) {
      System.out.println("New user: "+c);
      this.users.add(c);
      return 0;
   }

   @Override
   public int exit(ChatClient c) {
      System.out.println("User left: "+c);
      this.users.remove(c);
      return 0;
   }
```

Classe per oggetto remoto *server* (cont)

```
@Override
public int message(String message) {
   System.out.println("Received message: "+message);
   synchronized(this.users){
      for (Iterator<ChatClient> i= this.users.iterator(); i.hasNext();){
         ChatClient next= i.next();
         try {
            next.message(message);
         } catch (RemoteException e) {
            System.out.println("error forwarding to user: "
                  +next+"("+e.getMessage()+"). Removing from list.");
            i.remove();
         }
      }
   }
   return 0;
}

}
```

Classe per oggetto remoto cliente

```
public class PisaChatClient extends RemoteObject
implements ChatClient{

   private static final long serialVersionUID = -3099711356022216318L;

   @Override
   public void message(String message) {
      System.out.println(message);
   }

}
```

RMI registry deve essere stato avviato esternamente usando commando **`rmiregistry`** (**`start rmiregistry`** su Windows)

Avvio *server*

```
public class ChatRoomMain {

   public static void main(String[] args) {
      try{
         ChatRoom room= (ChatRoom) UnicastRemoteObject.
             exportObject(new PisaChatRoom(),0);
         Registry registry= LocateRegistry.getRegistry();
         registry.rebind(ChatRoom.OBJECT_NAME, room);
         System.out.println("Chat room ready");
      }catch(RemoteException e){
         System.out.println("Server error:" +e.getMessage());
      }
   }

}
```

Avvio clienti

```
public class ChatClientMain {

   public static void main(String[] args) {
      ChatClient user=new PisaChatClient();
      try {
         ChatClient userStub= (ChatClient) UnicastRemoteObject.
            exportObject(user,0);

         ChatRoom chatRoom= (ChatRoom) LocateRegistry.getRegistry().
            lookup(ChatRoom.OBJECT_NAME);

         chatRoom.enter(userStub);

      System.out.println("You are now logged in.
          Enter a new message or 'exit' to quit:");
      String userInput="";
      BufferedReader in =
         new BufferedReader(new InputStreamReader(System.in));
      while (!(userInput=in.readLine()).equals("exit")){
         chatRoom.message(userInput);
      }

      chatRoom.exit(user);
}
```

Avvio clienti

```
catch (RemoteException e) {
        System.out.println("Client error:" +e.getMessage());
    } catch (NotBoundException e) {
        System.out.println("Chat room not available :" +e.getMessage());
    } catch (UnsupportedEncodingException e) {
        System.out.println("Encoding not available :" +e.getMessage());
    } catch (IOException e) {
        System.out.println("Error reading user message :" +e.getMessage());
    } finally{
        try {
            UnicastRemoteObject.unexportObject(user, true);
        } catch (NoSuchObjectException e) {
            System.out.println("Counld not unexport :" +e.getMessage());
        }
    }
  }
}
```

```
Chat room ready
New user:
Proxy[ChatClient,RemoteObjectInvocationHandler[Unicast
Ref [liveRef: [endpoint:[192.168.1.73:57001]
(remote),objID:[706914f4:1545da8beae:-7fff,
-3626322549717799031]]]]]
Received message: hello
New user:
Proxy[ChatClient,RemoteObjectInvocationHandler[Unicast
Ref [liveRef: [endpoint:[192.168.1.73:57005]
(remote),objID:[6576e62f:1545da8ee9d:-7fff,
-338409369727429970]]]]]
Received message: hello, anybody there?
Received message: yes, it's me
Received message: me who?
Received message: me myself :)
Received message: ok, yourself
Received message: bye bye now
Received message: bye bye
User left:
Proxy[ChatClient,RemoteObjectInvocationHandler[Unicast
Ref [liveRef: [endpoint:[192.168.1.73:57001]
(remote),objID:[706914f4:1545da8beae:-7fff,
-3626322549717799031]]]]]
Received message: i am all alone now?
Received message: seems like it
User left:
Proxy[ChatClient,RemoteObjectInvocationHandler[Unicast
Ref [liveRef: [endpoint:[192.168.1.73:57005]
(remote),objID:[6576e62f:1545da8ee9d:-7fff,
-338409369727429970]]]]]
```

```
You are now logged in.
Enter a new message or
'exit' to quit:
hello
hello
hello, anybody there?
hello, anybody there?
yes, it's me
me who?
me who?
me myself :)
ok, yourself
ok, yourself
bye bye now
bye bye now
bye bye
exit
```

```
You are now logged in.
Enter a new message or
'exit' to quit:
hello, anybody there?
yes, it's me
yes, it's me
me who?
me myself :)
me myself :)
ok, yourself
bye bye now
bye bye
bye bye
i am all alone now?
i am all alone now?
seems like it
seems like it
exit
```

Conclusioni

- RMI - programmazione di rete ad alto livello
- Permette comunicazione cliente/*server* tramite metodi di oggetti
- Permette l'uso dei *callback*
- Oggi abbiamo ignorato tanti dettagli necessari per avviare client e server su macchine diverse (gli esempi che abbiamo discusso oggi funzionano solo sulla stessa macchina):
  - sicurezza
  - *code mobility*
  - *dynamic class loading*

Esercizi

1. Gestione congresso

- Si progetti un'applicazione Client/*Server* per la gestione delle registrazioni ad un congresso.

- L'organizzazione del congresso fornisce agli *speaker* delle varie sessioni un'interfaccia tramite la quale iscriversi ad una sessione, e la possibilità di visionare i programmi delle varie giornate del congresso, con gli interventi delle varie sessioni.

- Il *server* mantiene i programmi delle 3 giornate del congresso, ciascuno dei quali è memorizzato in una struttura dati, in cui ad ogni elemento corrisponde una sessione (in tutto 12 per ogni giornata). Per ciascuna sessione vengono memorizzati i nomi degli *speaker* che si sono registrati (al massimo 5).

Esercizi

- Il cliente può richiedere operazioni per
  - registrare uno *speaker* ad una sessione;
  - ottenere il programma del congresso;
- Il cliente inoltra le richieste al *server* tramite il meccanismo di RMI.
- Prevedere, per ogni possibile operazione una gestione di eventuali condizioni anomale (ad esempio la richiesta di registrazione ad una giornata e/o sessione inesistente oppure per la quale sono già stati coperti tutti gli spazi d'intervento).
- Il client è implementato come un processo ciclico che continua a fare richieste sincrone fino ad esaurire tutte le esigenze utente. Stabilire una opportuna condizione di terminazione del processo di richiesta.

Esercizi

2. Gestione forum (prossima settimana)

- Un *forum* è caratterizzato da un argomento su cui diversi utenti possono scambiarsi opinioni via rete. Il sistema deve prevedere un *server* RMI che fornisca le seguenti funzionalità:

  - apertura di un nuovo *forum*, di cui è specificato l'argomento (esempio: giardinaggio)?

  - inserimento di un nuovo messaggio indirizzato ad un *forum* identificato dall'argomento (es: è tempo di piantare le viole, per il forum giardinaggio)?

  - registrazione di una *callback* da parte dell'utente. La *callback* contiene un riferimento ad uno dei forum attivi. Il server notifica al client, mediante la *callback* registrata , l'invio di ogni nuovo messaggio a quel forum.