

Java Remote Method Invocation - 2



Corso di Laurea in Informatica, Programmazione Distribuita
 Delfina Malandrino, dmalandrino@unisa.it
<http://www.unisa.it/docenti/delfinamalandrino>

1

Organizzazione della lezione

2

- L'architettura a layer di Java RMI
 - ▣ Il meccanismo di marshalling
- Il processo di creazione
- Il primo esempio: HelloWorld!
- Conclusioni

2

Somiglianza fra oggetti locali e remoti

3

- Per facilitare la programmabilità dell'ambiente
 - ▣ familiare e tradizionale paradigma
- Questo non significa totale trasparenza
 - ▣ il programmatore deve sapere cosa è locale e cosa è distribuito
 - ▣ per poter comprendere le differenze nella semantica ad esempio, quello del passaggio di parametri
- RemoteObject ridefinisce metodi di Object
 - ▣ nel caso in cui si usi la implementazione locale (exportObject()) questi metodi vanno ridefiniti a carico del programmatore

3

Metodi ridefiniti da RemoteObject

4

- X.hashCode()
 - ▣ restituisce lo stesso codice per due stub diversi che si riferiscono allo stesso oggetto remoto
 - ▣ utilizzo come chiavi nelle tabelle hash
- X.equals()
 - ▣ eguaglianza sugli stub (non sugli oggetti remoti)
 - ▣ impossibile effettuare controllo su oggetti remoti
 - sarebbe necessario che equals() lanci RemoteException
 - ▣ . . . ma anche molto inefficiente
- toString()
 - ▣ restituisce informazioni sulla macchina che ospita l'oggetto oltre che tradizionale hash e nome della classe

4

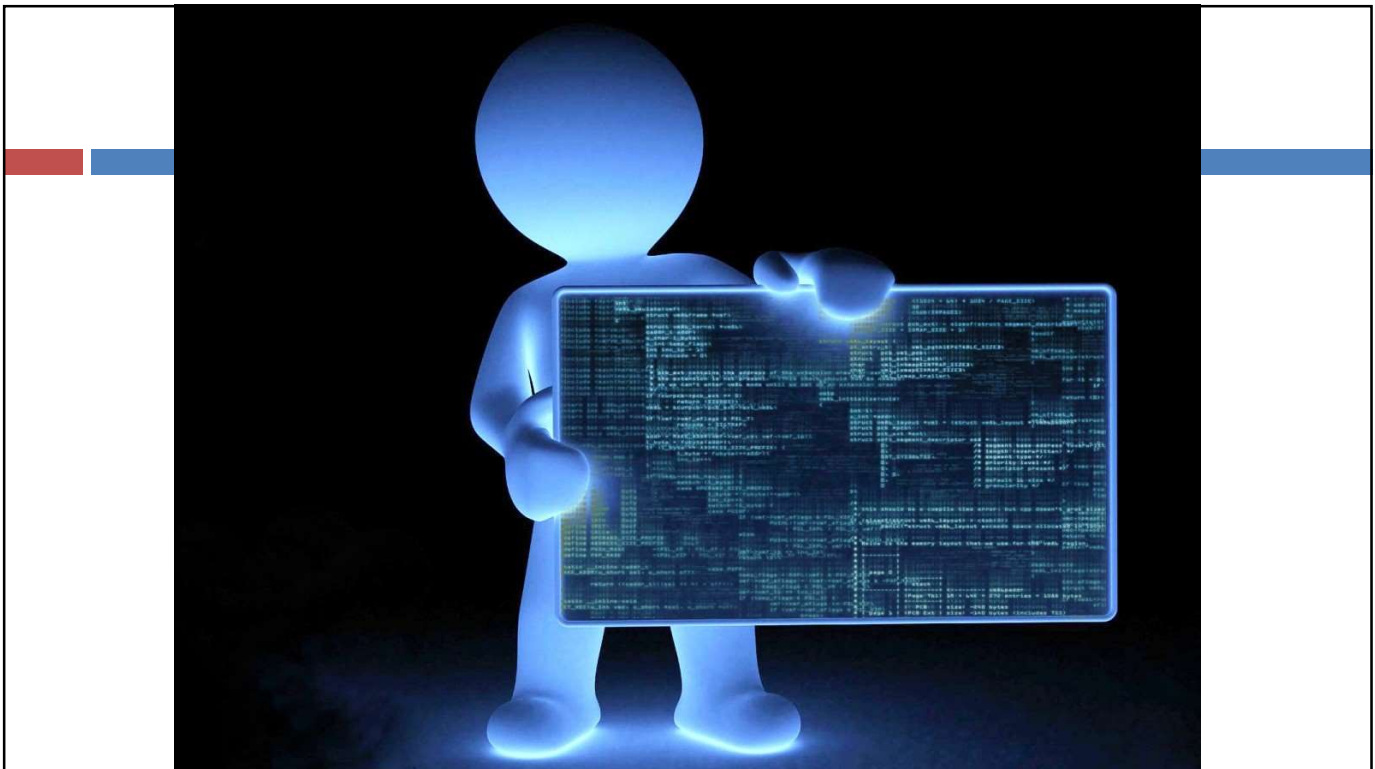
Altre differenze...

5

- Passaggio di parametri remoti
 - ▣ passati come parametri o restituiti
 - ▣ per riferimento (quindi con semantica tradizionale Java)
- Gestione di tipi
 - ▣ uso di casting Java
 - ▣ uso di instanceof sul riferimento remoto per verificare le interfacce supportate da un oggetto remoto
- Compilatore forza a trattare le RemoteException
 - ▣ come checked exception
 - ▣ il programmatore è conscio delle invocazioni remote



5



6

Organizzazione della lezione

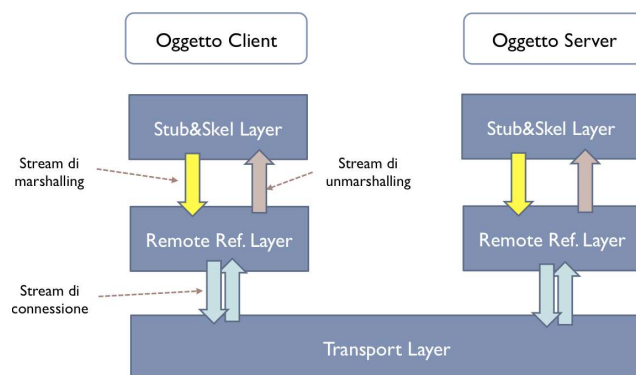
7

- L'architettura a layer di Java RMI
 - ▣ Il meccanismo di marshalling
- Il processo di creazione
- Il primo esempio: HelloWorld!
- Conclusioni

7

I layer dell'architettura

8

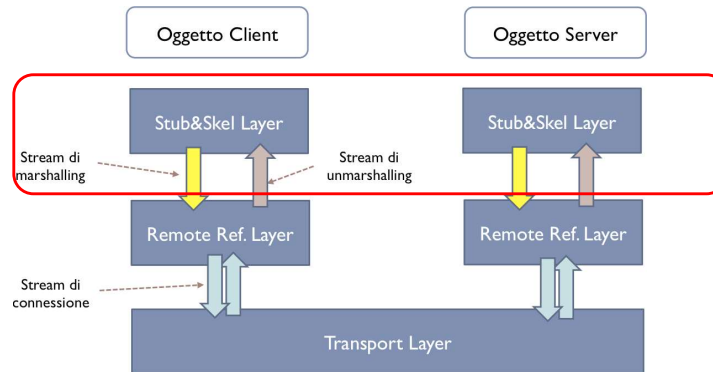


- Astrazione su tre livelli
- Permette evoluzione separata dei layer

8

Stub & Skeleton

9



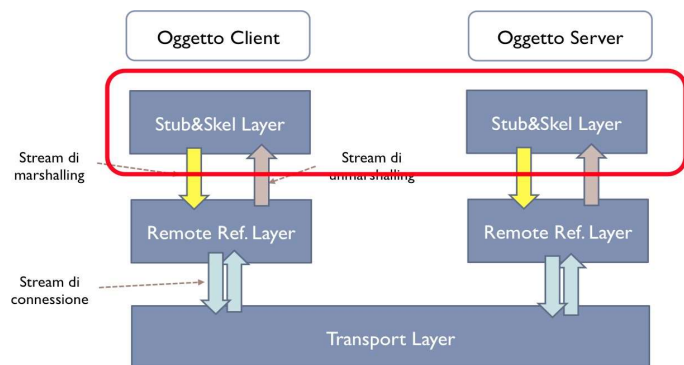
- Interfaccia tra l'applicazione (le classi scritte dal programmatore) ed il resto del sistema

9

Cosa fa lo Stub & Skeleton Layer - 1

10

- Verso il basso (Remote Reference Layer)
 - ▣ fornisce uno stream di marshal di oggetti (passati per copia)
- I compiti dello Stub
 - ▣ iniziare la connessione con la macchina virtuale remota attraverso il Remote Reference layer
 - ▣ mashalling di oggetti verso il RRL
 - ▣ attende il risultato della invocazione
 - ▣ effettua l'unmarshalling degli oggetti
 - ▣ restituisce risultato a oggetto client

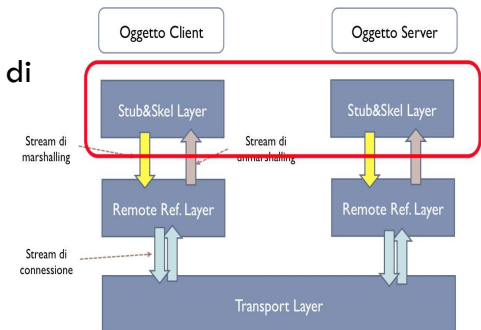


10

Cosa fa lo Stub & Skeleton Layer - 2

11

- Lo skeleton quando riceve una invocazione effettua l'unmarshalling dal RRL dei parametri invoca il metodo remoto
 - ▣ effettua il marshalling del valore restituito
- Creazione di stub e compiler punto critico della facilità di uso di Java RMI
 - ▣ `rmic`, tool di JDK, genera a partire dalla classe dell'oggetto remoto
 - ▣ ora non più necessario l'uso
- Le evoluzioni del layer durante le varie versioni
 - ▣ skeleton eliminato da Java 2, con codice generico (uso di reflection)
 - ▣ stub generati in maniera dinamica (in maniera nascosta al programmatore) tramite un Proxy

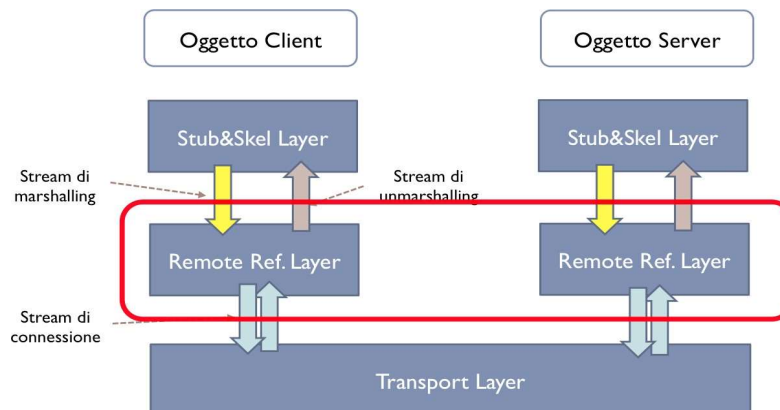


11

Remote Reference Layer

12

- Interfaccia lo Stub & Skeleton Layer verso il trasporto e si occupa della semantica della invocazione

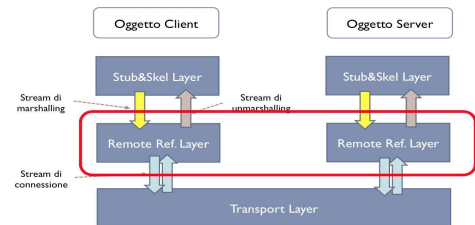


12

Gli obiettivi di questo layer

13

- Diverse possibilità per il comportamento della invocazione remota
 - ▣ invocazione unicast
 - ▣ invocazione multicast ad un insieme di server replicati (ridondanza e tolleranza ai malfunzionamenti)
 - ▣ oggetti attivabili (presenti in memoria secondaria e richiamati all'atto dell'invocazione)
 - ▣ riconnessione automatica
- Solo unicast (1.1) e oggetti attivabili (1.2) sono forniti in RMI
- Altre funzionalità assorbite da "strati" di software superiore al middleware di trasporto

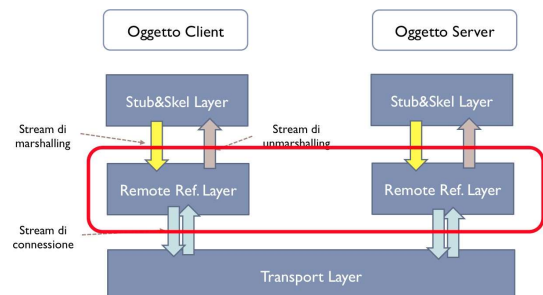


13

Come funziona lo strato

14

- Il layer fornisce verso lo Stub&Skeleton Layer una interfaccia `java.rmi.server.RemoteServer`
 - ▣ che espone un metodo `invoke`
- Esempio degli aspetti positivi della astrazione
 - ▣ l'eliminazione dello skeleton resa possibile dalla modifica di questo layer (`RemoteServer`)
- Verso il basso (Transport Layer) usa lo stream di connessione

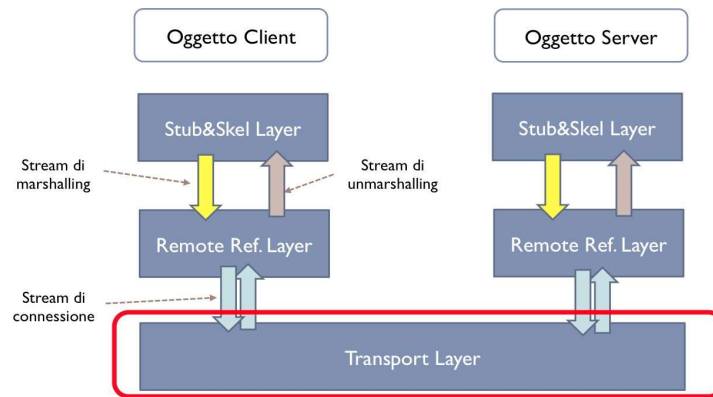


14

Transport Layer

15

- Layer che offre comunicazione connection-oriented (possibile l'implementazione interna connectionless)

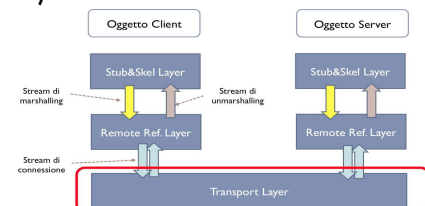


15

Cosa fa questo layer

16

- Connessione con macchine con IP remoti
 - ▣ gestione delle connessioni
 - ▣ attesa di connessioni in arrivo
 - ▣ stabilire connessioni per le chiamate in entrata
- Gestione di una tabella di oggetti remoti che sono attivi nella JVM
 - ▣ identificazione dell'oggetto dispatcher a cui passare la comunicazione
- Java Remote Method Protocol
- Implementazione con Internet Inter-ORB Protocol (RMI-IIOP)
 - ▣ dal mondo CORBA
 - ▣ utilizzato in Java Enterprise



16

Organizzazione della lezione

17

- L'architettura a layer di Java RMI
 - ▣ Il meccanismo di marshalling
- Il processo di creazione
- Il primo esempio: HelloWorld!
- Conclusioni

17

Marshalling vs. Serializzazione

18

- La serializzazione di Java trasforma un oggetto in uno stream di byte
 - ▣ utilizzata per vari scopi oltre che per rete (scrittura su file, etc)
- Il Marshalling aggiunge al flusso di serializzazione effettuando delle modifiche
 - ▣ modifica della semantica degli oggetti remoti
 - ▣ informazioni aggiuntive sull'oggetto
- Specializzazione di ObjectOutputStream
 - ▣ attraverso la ridefinizione di alcuni metodi

18

Specializzazione di ObjectOutputStream

19

- **replaceObject()**
 - ▣ se l'oggetto da serializzare è una istanza di `java.rmi.Remote()`
 - quindi un riferimento ad un oggetto remoto
 - ▣ allora un riferimento ad un oggetto remoto viene sostituito dal suo stub (usando il metodo `RemoteObject.toStub()`)
 - ▣ attivata da un flag `enableReplaceObject`
 - ▣ integrità referenziale assicurata
 - ▣ nota: riferimenti remoti non sono serializzabili. . . ma `RemoteStub` lo è
- **annotateClass()**
 - ▣ informazioni aggiuntive sulla posizione della classe da caricare
 - ▣ caricamento dinamico della classe

19

Organizzazione della lezione

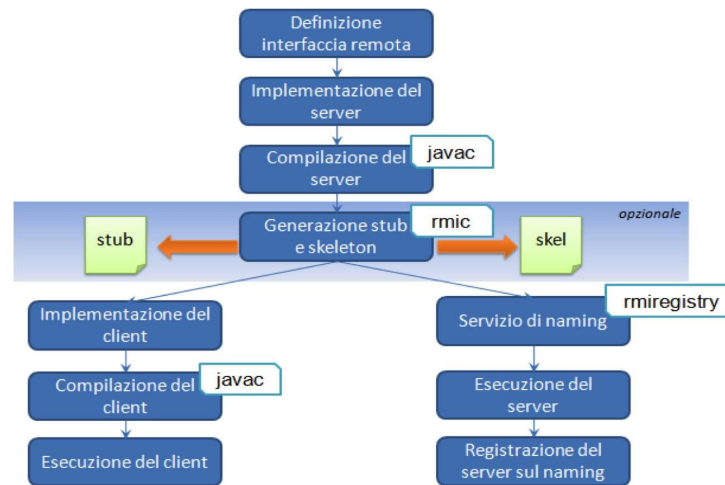
20

- Differenze tra oggetti locali e remoti
- L'architettura a layer di Java RMI
 - ▣ Il meccanismo di marshalling
- **Il processo di creazione**
- Il primo esempio: `HelloWorld!`
- Conclusioni

20

Una visione d'insieme

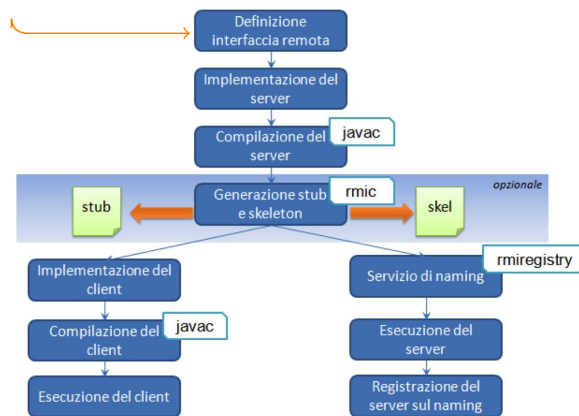
21



21

1: DEFINIZIONE DELL'INTERFACCIA REMOTA

22

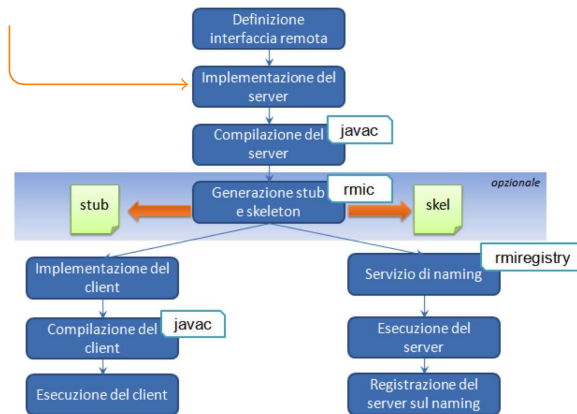


Specifica dei servizi offerti dal server
 Interface deve derivare da `Remote`
 Tutti i metodi remoti devono lanciare la eccezione
`java.rmi.RemoteException`

22

2: IMPLEMENTAZIONE DEL SERVER

23



Oggetto remoto istanza di una classe che implementa una o più interfacce remote

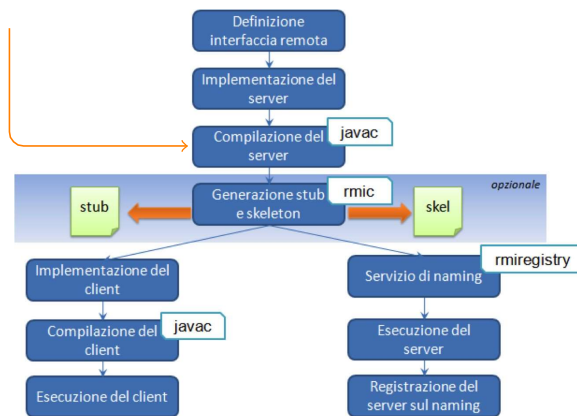
Deriva da `Unicast-RemoteObject`

Costruttore deve lanciare `RemoteException`

23

3: COMPILAZIONE DEL SERVER

24



Generazione dei file

.class

Posizione dei file importante

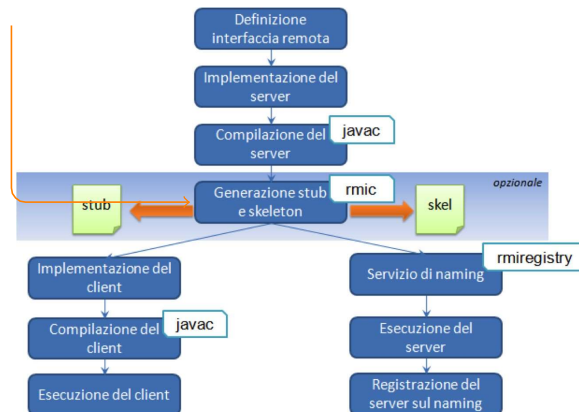
Usare la view apposita nella IDE

ad esempio,
la Navigator
in Eclipse

24

4: GENERAZIONE STUB E SKELETON

26



Generazione ora è automatica

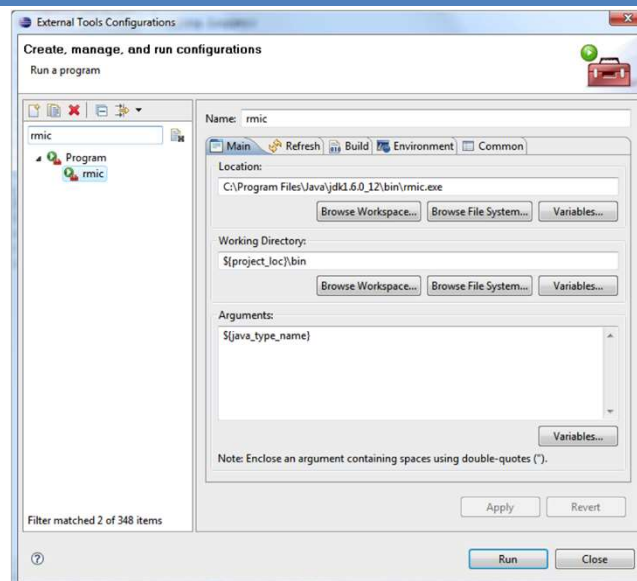
Uso dello stub compiler rmic
rmic nomeclasse

Non più necessario da JDK 1.1 e 1.2, ma per motivi di efficienza a volte utile

25

RMIC COME APPLICAZIONE ESTERNA

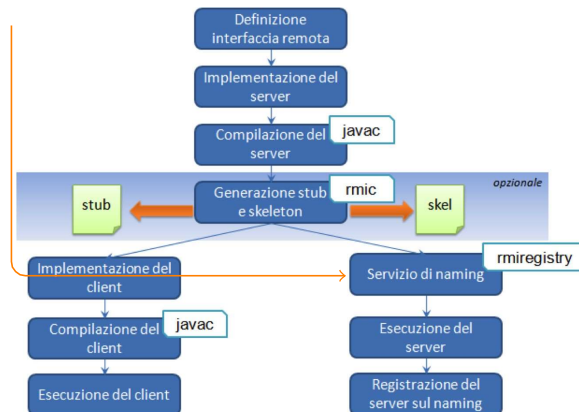
27



26

5 (SERVER): SERVIZIO DI NAMING

27



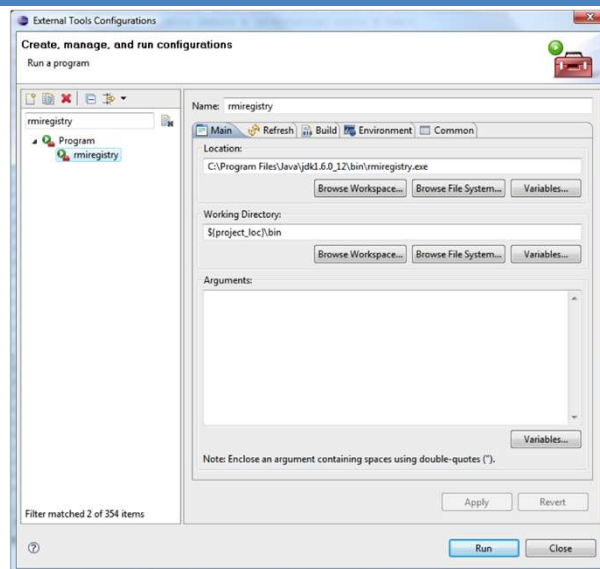
Semplice servizio di naming

Deve essere lanciato dalla directory dove si trova il file .class di oggetto remoto, stub e interfaccia remota

27

RMIREGISTRY COME APPLICAZIONE ESTERNA

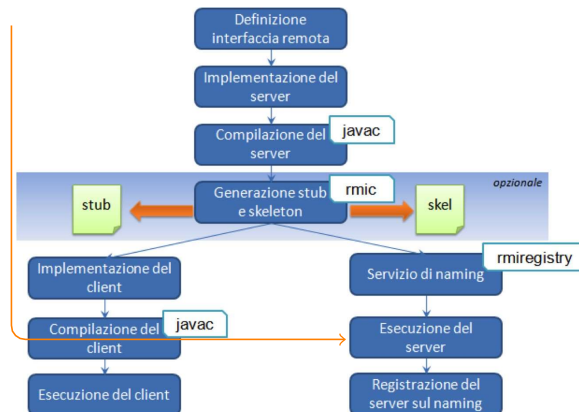
28



28

6: ESECUZIONE DEL SERVER

29



Esecuzione con JVM
 Specifica della politica di sicurezza attraverso un file di policy

29

Il file di policy

30

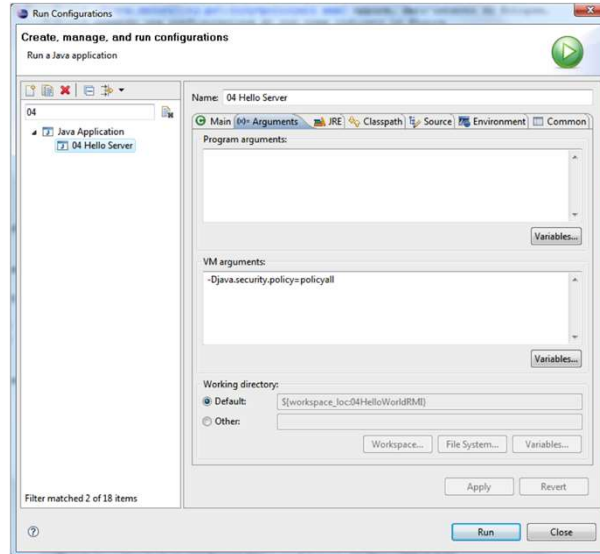
- Policy utilizzata dal Security Manager per stabilire i permessi da attribuire a ciascuna azione potenzialmente pericolosa
- Eseguire la macchina virtuale con
 - ▣ `java -Djava.security.policy=nomefile`
- Negli esempi, usiamo un file di massima apertura
 - ▣ attenzione, da non seguire nelle applicazioni reali!
- Deve trovarsi nella directory da dove lanciamo l'applicazione
 - ▣ se non si trova, silenziosamente non viene usata nessuna policy

```
grant {
    permission java.security.AllPermission;
};
```

30

LANCIO DEL SERVER

31



31

Cosa succede senza file di policy

32

```

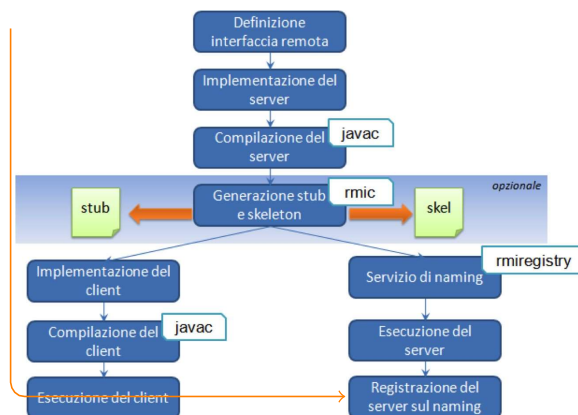
24-gen-2009 15:28:57 HelloImpl main
INFO: Crea l'oggetto remoto...
24-gen-2009 15:29:00 HelloImpl main
INFO: ... ora ne effettuo il rebind...
java.security.AccessControlException: access denied (java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
    at java.security.AccessControlContext.checkPermission(Unknown Source)
    at java.security.AccessController.checkPermission(Unknown Source)
    at java.lang.SecurityManager.checkPermission(Unknown Source)
    at java.lang.SecurityManager.checkConnect(Unknown Source)
    at java.net.Socket.connect(Unknown Source)
    at java.net.Socket.connect(Unknown Source)
    at java.net.Socket.<init>(Unknown Source)
    at java.net.Socket.<init>(Unknown Source)
    at sun.rmi.transport.proxy.RMIDirectSocketFactory.createSocket(Unknown Source)
    at sun.rmi.transport.proxy.RMIMasterSocketFactory.createSocket(Unknown Source)
    at sun.rmi.transport.tcp.TCPEndpoint.newSocket(Unknown Source)
    at sun.rmi.transport.tcp.TCPChannel.createConnection(Unknown Source)
    at sun.rmi.transport.tcp.TCPChannel.newConnection(Unknown Source)
    at sun.rmi.server.UnicastRef.newCall(Unknown Source)
    at sun.rmi.registry.RegistryImpl_Stub.rebind(Unknown Source)

```

32

7: REGISTRAZIONE SERVER SU SERVIZIO DI NAMING

33



Uso dei metodi statici di
`java.rmi.Naming`

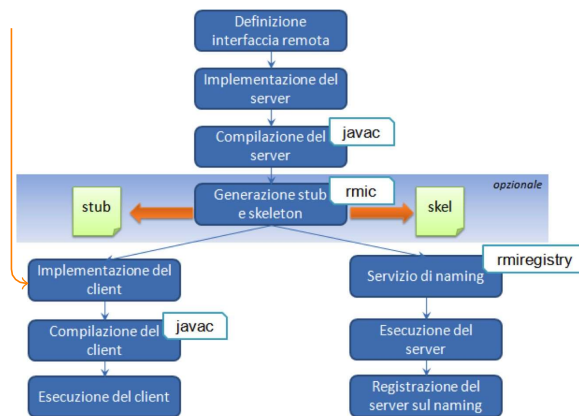
Per sicurezza, naming di
`rmiregistry` sullo stesso host del
server

limitazione notevole per servizi
distribuiti

33

8 (CLIENT): IMPLEMENTAZIONE DEL CLIENT

34



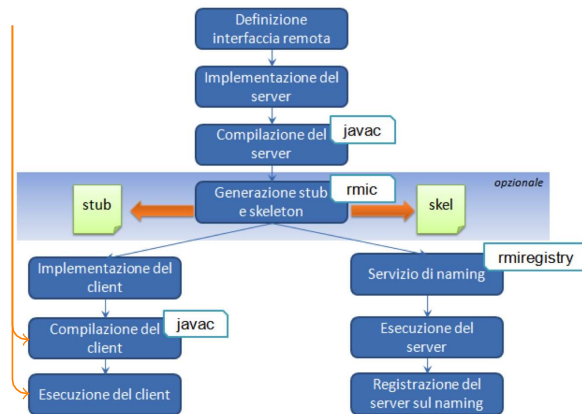
Uso dei metodi statici di
`java.rmi.Naming` per accedere
all'oggetto remoto

34

9-10: COMPILAZIONE E ESECUZIONE

35

Standard



35

Organizzazione della lezione

36

- ☐ Differenze tra oggetti locali e remoti
- ☐ L'architettura a layer di Java RMI
 - ☐ Il meccanismo di marshalling
- ☐ Il processo di creazione
- ☒ Il primo esempio: HelloWorld!
- ☐ Conclusioni

36

Primo programma RMI

37

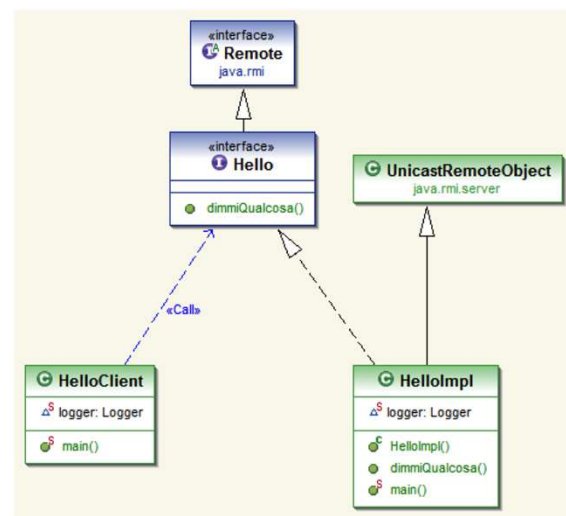
- Semplice HelloWorld (secondo la tradizione)
- Per ora, scritto in un solo “progetto”
 - ▣ questo rappresenta una forzatura: client e server hanno evoluzioni diverse e hanno bisogno di progetti diversi
- Obiettivo: il client invia il suo nome al server, che lo saluta,
 - ▣ stampando a video il nome del client
 - ▣ rimanendo in attesa di successivi client

37

Diagramma delle classi

38

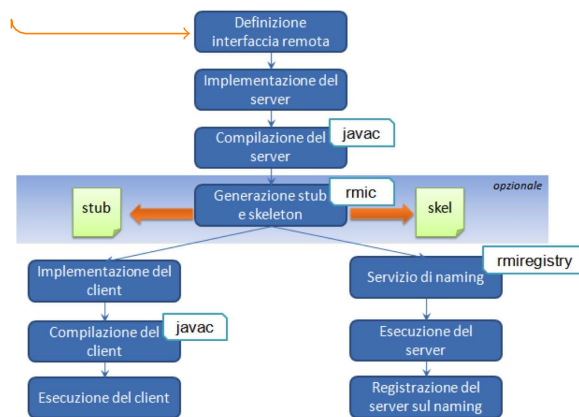
- Semplice HelloWorld (secondo la tradizione)



38

1: DEFINIZIONE DELL'INTERFACCIA REMOTA

39

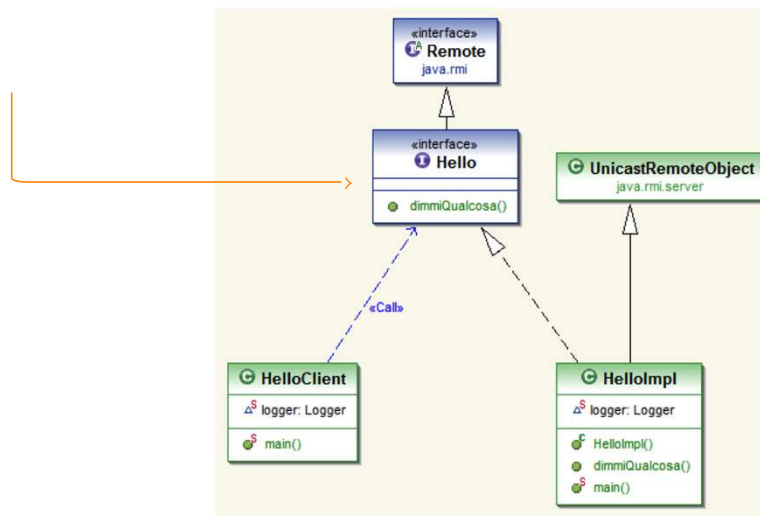


Specifica dei servizi offerti dal server
Interface deve derivare da Remote
Tutti i metodi remoti devono lanciare la eccezione
`java.rmi.RemoteException`

39

Il Diagramma delle classi: l'interfaccia

40



40

L'interfaccia Hello

41

```
public interface Hello {  
    extends java.rmi.Remote {  
        String dimmiQualcosa(String daChi)  
            throws java.rmi.RemoteException;  
    }  
}
```

Dichiarazione interfaccia

41

L'interfaccia Hello

42

```
public interface Hello  
    extends java.rmi.Remote {  
        String dimmiQualcosa(String daChi)  
            throws java.rmi.RemoteException;  
    }  
}
```

Dichiarazione interfaccia

Estende la interfaccia marker

42

L'interfaccia Hello

43

```
public interface Hello
    extends java.rmi.Remote {
    String dimmiQualcosa(String daChi)
        throws java.rmi.RemoteException;
}
```

Dichiarazione interfaccia

Estende la interfaccia marker

Unico metodo dichiarato, con
parametro

43

L'interfaccia Hello

44

```
public interface Hello
    extends java.rmi.Remote {
    String dimmiQualcosa(String daChi)
        throws java.rmi.RemoteException;
}
```

Dichiarazione interfaccia

Estende la interfaccia marker

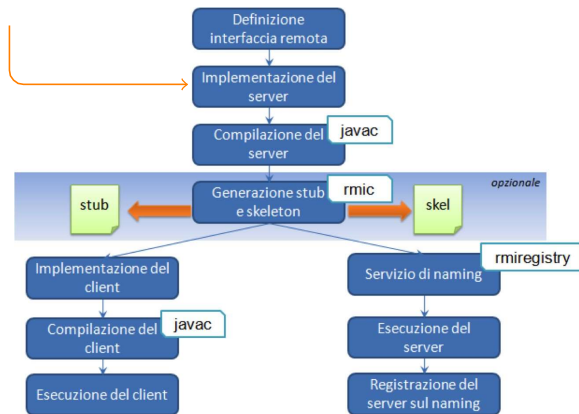
Unico metodo dichiarato, con
parametro

Metodo remoto: lancia eccezione
remota

44

2: IMPLEMENTAZIONE DEL SERVER

45



Oggetto remoto istanza di una classe che implementa una o più interfacce remote

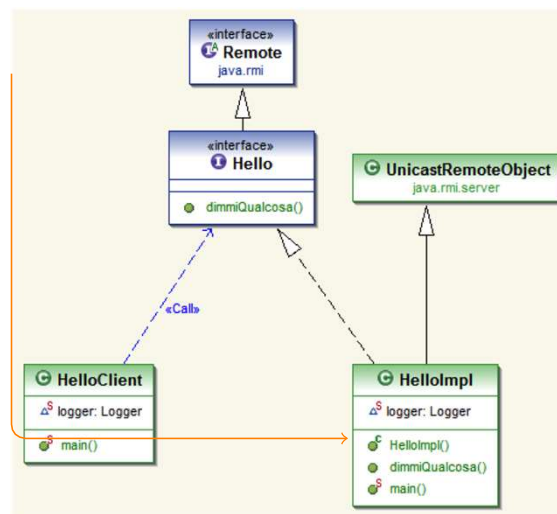
Deriva da `UnicastRemoteObject`

Costruttore deve lanciare `RemoteException`

45

Il Diagramma delle classi: il server

46



46

La classe HelloImpl

47

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

47

La classe HelloImpl

48

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

Estende la classe di RMI

48

La classe HelloImpl

49

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

Estende la classe di RMI

...implementa l'interfaccia remota

49

La classe HelloImpl

50

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

Estende la classe di RMI

... implementa l'interfaccia remota

Serial per la versione

50

La classe HelloImpl

51

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

Estende la classe di RMI

...implementa l'interfaccia remota

Serial per la versione

Costruttore vuoto (con lancio
eccezione)

51

La classe HelloImpl

52

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

Estende la classe di RMI

...implementa l'interfaccia remota

Serial per la versione

Costruttore vuoto (con lancio
eccezione)

Metodo remoto

52

La classe HelloImpl

53

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

Estende la classe di RMI

...implementa l'interfaccia remota

Serial per la versione

Costruttore vuoto (con lancio
eccezione)

Metodo remoto

Log locale

53

La classe HelloImpl

54

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class HelloImpl
    extends UnicastRemoteObject
    implements Hello {

    private static final long serialVersionUID =
        -4469091140865645865L;
    static Logger logger= Logger.getLogger("global");

    public HelloImpl() throws RemoteException {
        //vuoto
    }

    public String dimmiQualcosa(String daChi)
        throws RemoteException {
        logger.info("Sto salutando"+daChi);
        return "Ciao!";
    }

    //..
}
```

Import

Estende la classe di RMI

...implementa l'interfaccia remota

Serial per la versione

Costruttore vuoto (con lancio
eccezione)

Metodo remoto

Log locale

Stampa remota

54

La classe HelloImpl

55

```
//...
public static void main(String args[]) {
    System.setSecurityManager(new
        RMISecurityManager());

    try{
        logger.info("Crea l'oggetto remoto...");
        HelloImpl obj = new HelloImpl();
        logger.info("...ne effettuo il rebind...");
        Naming.rebind("HelloServer", obj);
        logger.info("...Pronto!");
    } catch (Exception e) {
        e.printStackTrace();
    }

} //end main
} //end classe HelloImpl
```

Metodo thread main

55

La classe HelloImpl

56

```
//...
public static void main(String args[]) {
    System.setSecurityManager(new
        RMISecurityManager());

    try{
        logger.info("Crea l'oggetto remoto...");
        HelloImpl obj = new HelloImpl();
        logger.info("...ne effettuo il rebind...");
        Naming.rebind("HelloServer", obj);
        logger.info("...Pronto!");
    } catch (Exception e) {
        e.printStackTrace();
    }

} //end main
} //end classe HelloImpl
```

Metodo thread main

Security manager

56

La classe HelloImpl

57

```
//...
public static void main(String args[]) {
    System.setSecurityManager(new
        RMISecurityManager());

    try{
        logger.info("Creo l'oggetto remoto...");
        HelloImpl obj =new HelloImpl();
        logger.info("...ne effettuo il rebind...");
        Naming.rebind("HelloServer", obj);
        logger.info("...Pronto!");
    }catch (Exception e) {
        e.printStackTrace();
    }

} //end main
} //end classe HelloImpl
```

Metodo thread main

Security manager

Si crea l'oggetto remoto

57

La classe HelloImpl

58

```
//...
public static void main(String args[]) {
    System.setSecurityManager(new
        RMISecurityManager());

    try{
        logger.info("Creo l'oggetto remoto...");
        HelloImpl obj =new HelloImpl();
        logger.info("...ne effettuo il rebind...");
        Naming.rebind("HelloServer", obj);
        logger.info("...Pronto!");
    }catch (Exception e) {
        e.printStackTrace();
    }

} //end main
} //end classe HelloImpl
```

Metodo thread main

Security manager

Si crea l'oggetto remoto

Se ne fa rebind sul rmiregistry

58

La classe HelloImpl

59

```
//...
public static void main(String args[]) {
    System.setSecurityManager(new
        RMISecurityManager());

    try{
        logger.info("Creo l'oggetto remoto...");
        HelloImpl obj =new HelloImpl();
        logger.info("...ne effettuo il rebind...");
        Naming.rebind("HelloServer", obj);
        logger.info("...Pronto!");
    }catch(Exception e) {
        e.printStackTrace();
    }

} //end main
} //end classe HelloImpl
```

Metodo thread main

Security manager

Si crea l'oggetto remoto

Se ne fa rebind sul rmiregistry

Blocco try...catch

59

La classe HelloImpl

60

```
//...
public static void main(String args[]) {
    System.setSecurityManager(new
        RMISecurityManager());

    try{
        logger.info("Creo l'oggetto remoto...");
        HelloImpl obj =new HelloImpl();
        logger.info("...ne effettuo il rebind...");
        Naming.rebind("HelloServer", obj);
        logger.info("...Pronto!");
    }catch(Exception e) {
        e.printStackTrace();
    }

} //end main
} //end classe HelloImpl
```

Metodo thread main

Security manager

Si crea l'oggetto remoto

Se ne fa rebind sul rmiregistry

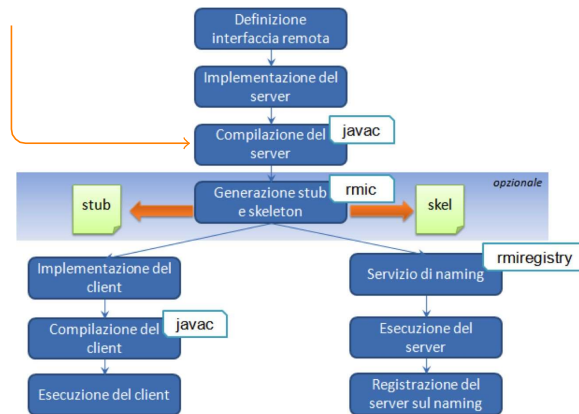
Blocco try...catch

Fine del main (thread in esecuzione)

60

3: COMPILAZIONE DEL SERVER

61



Generazione dei file
.class

Posizione dei file
importante

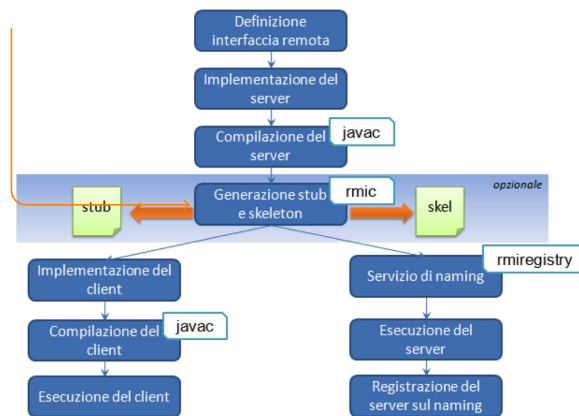
Usare la view
apposita nella IDE

ad esempio, la
Navigator in
Eclipse

61

4: GENERAZIONE STUB E SKELETON

62



Generazione ora è
automatica

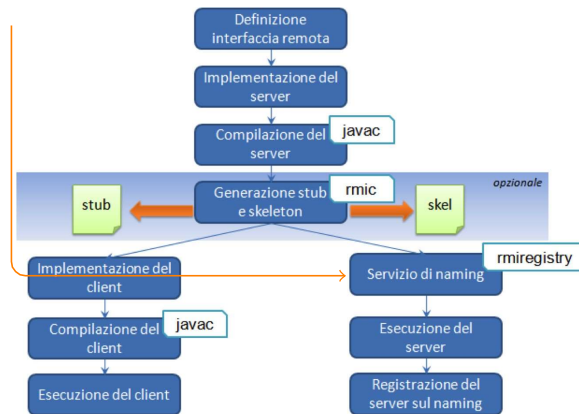
Uso dello stub
compiler rmic
rmic nomeclasse

Non più necessario
da JDK 1.1 e 1.2, ma
per motivi di
efficienza a volte
utile

62

5 (SERVER): SERVIZIO DI NAMING

63



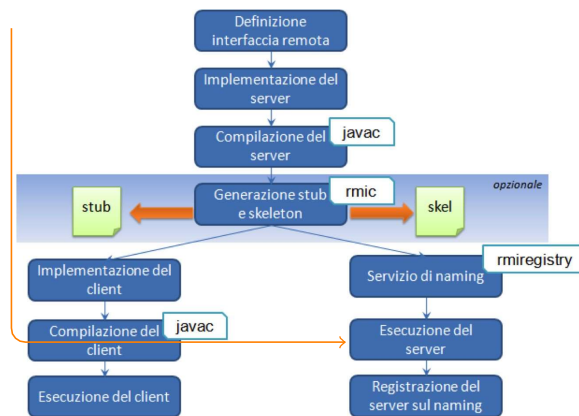
Semplice servizio di naming

Deve essere lanciato dalla directory dove si trova il file .class di oggetto remoto, stub e interfaccia remota

63

6: ESECUZIONE DEL SERVER

64

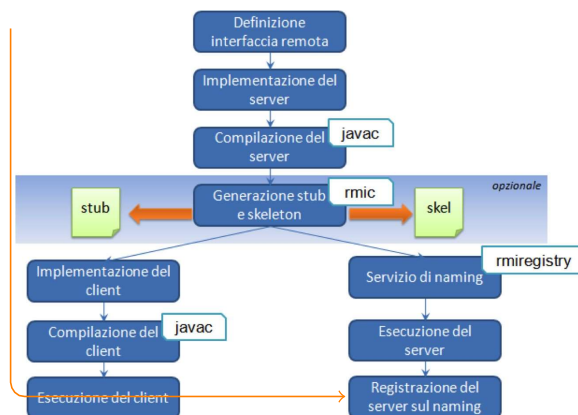


Esecuzione con JVM
Specifica della politica di sicurezza attraverso un file di policy

64

7: REGISTRAZIONE SERVER SU SERVIZIO DI NAMING

65



Uso dei metodi statici di
`java.rmi.Naming`

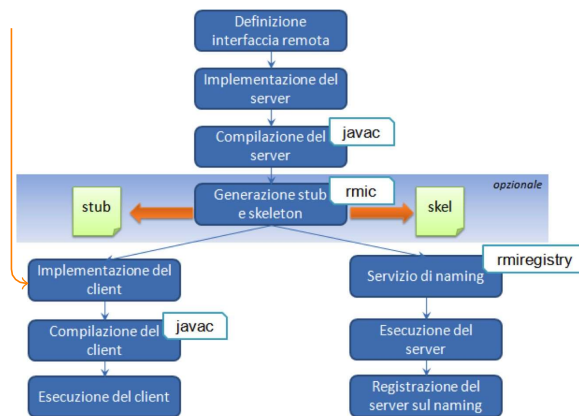
Per sicurezza, naming di
`rmiregistry` sullo stesso host del
server

limitazione notevole per servizi
distribuiti

65

8 (CLIENT): IMPLEMENTAZIONE DEL CLIENT

66

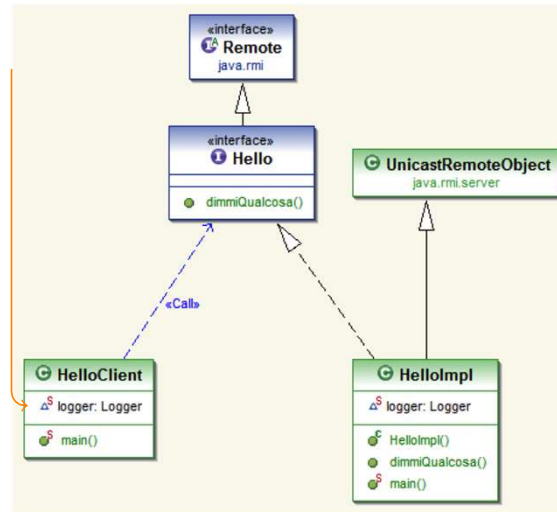


Uso dei metodi statici di
`java.rmi.Naming` per accedere
all'oggetto remoto

66

Il Diagramma delle classi: Il Client

67



67

La classe HelloClient

68

```

import java.rmi. ← Import
import java.util.logging.Logger;

public class HelloClient {
    static Logger logger= Logger.getLogger("global");

    public static void main(String args[]) {
        try{
            logger.info("Sto cercando l'oggetto remoto...");
            Hello obj = (Hello)
                Naming.lookup("rmi://localhost/HelloServer");

            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");

            System.out.println("Ricevuto:"+ risultato);
        }catch(Exception e) {
            e.printStackTrace();
        }
    } //fine main
} //fine classe HelloClient
  
```

68

La classe HelloClient

69

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

public class HelloClient {
    static Logger logger= Logger.getLogger("global");

    public static void main(String args[]) {
        try{
            logger.info("Sto cercando l'oggetto remoto...");
            Hello obj = (Hello)
                Naming.lookup("rmi://localhost/HelloServer");

            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");

            System.out.println("Ricevuto:"+ risultato);
        } catch (Exception e) {
            e.printStackTrace();
        }
    } //fine main
} //fine classe HelloClient
```

Import

Classe

69

La classe HelloClient

70

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

public class HelloClient {
    static Logger logger= Logger.getLogger("global");

    public static void main(String args[]) {
        try{
            logger.info("Sto cercando l'oggetto remoto...");
            Hello obj = (Hello)
                Naming.lookup("rmi://localhost/HelloServer");

            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");

            System.out.println("Ricevuto:"+ risultato);
        } catch (Exception e) {
            e.printStackTrace();
        }
    } //fine main
} //fine classe HelloClient
```

Import

Classe

Metodo main

70

La classe HelloClient

71

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

public class HelloClient {
    static Logger logger= Logger.getLogger("global");

    public static void main(Strinargs[]) {
        try{
            logger.info("Sto cercando l'oggetto remoto...");
            Hello obj = (Hello)
                Naming.lookup("rmi://localhost/HelloServer");

            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");

            System.out.println("Ricevuto:"+ risultato);
        }catch (Exception e) {
            e.printStackTrace();
        }
    } //fine main
} //fine classe HelloClient
```

Import

Classe

Metodo main

Si cerca il riferimento remoto...

71

La classe HelloClient

72

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

public class HelloClient {
    static Logger logger= Logger.getLogger("global");

    public static void main(String args[]) {
        try{
            logger.info("Sto cercando l'oggetto remoto...");
            Hello obj = (Hello)
                Naming.lookup("rmi://localhost/HelloServer");

            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");

            System.out.println("Ricevuto:"+ risultato);
        }catch (Exception e) {
            e.printStackTrace();
        }
    } //fine main
} //fine classe HelloClient
```

Import

Classe

Metodo main

Si cerca il riferimento remoto...

...e se ne fa il casting (Java è strongly typed)

72

La classe HelloClient

73

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

public class HelloClient {
    static Logger logger= Logger.getLogger("global");

    public static void main(String args[]) {
        try{
            logger.info("Sto cercando l'oggetto remoto...");
            Hello obj = (Hello)
                Naming.lookup("rmi://localhost/HelloServer");

            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");

            System.out.println("Ricevuto:"+ risultato);
        } catch (Exception e) {
            e.printStackTrace();
        }
    } //fine main
} //fine classe HelloClient
```

Import

Classe

Metodo main

Si cerca il riferimento remoto...

...e se ne fa il casting (Java è *strongly typed*)

Invocazione remota

73

La classe HelloClient

74

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

public class HelloClient {
    static Logger logger= Logger.getLogger("global");

    public static void main(String args[]) {
        try{
            logger.info("Sto cercando l'oggetto remoto...");
            Hello obj = (Hello)
                Naming.lookup("rmi://localhost/HelloServer");

            logger.info("...Trovato! Invoco metodo...");
            String risultato = obj.dimmiQualcosa("Pippo");

            System.out.println("Ricevuto:"+ risultato);
        } catch (Exception e) {
            e.printStackTrace();
        }
    } //fine main
} //fine classe HelloClient
```

Import Classe

Metodo main

Si cerca il riferimento remoto...

...e se ne fa il casting (Java è *strongly typed*)

Invocazione remota

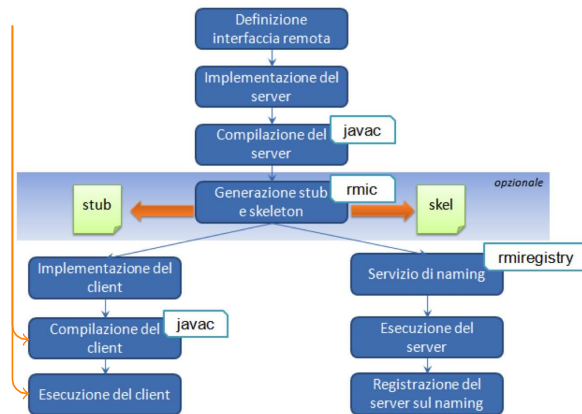
Stampa risultato

74

9-10: COMPILAZIONE E ESECUZIONE

75

Standard



75

L'ESECUZIONE: RMIREGISTRY

76

```

C:\WINDOWS\SYSTEM32\cmd.exe - rmiregistry

C:\Documents and Settings\vitsca\Desktop\Didattica\ProgDist\Progetti\HelloServer
>rmiregistry
  
```

76

L'ESECUZIONE: SERVER

77



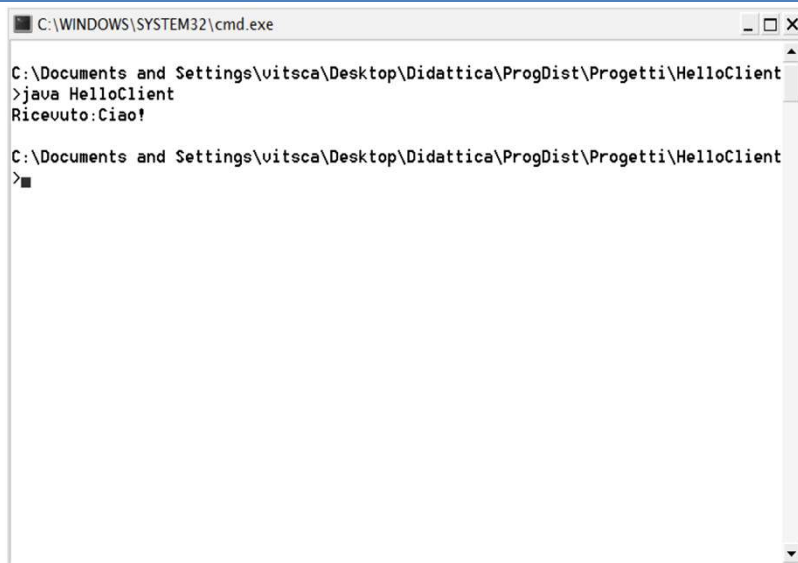
```
C:\WINDOWS\system32\cmd.exe - java -Djava.security.policy=policyall HelloImpl

C:\Documents and Settings\vitsca\Desktop\Didattica\ProgDist\Progetti\HelloServer
>java -Djava.security.policy=policyall HelloImpl
Effettuo il rebind...
Pronto!
■
```

77

L'ESECUZIONE: CLIENT

78



```
C:\WINDOWS\SYSTEM32\cmd.exe

C:\Documents and Settings\vitsca\Desktop\Didattica\ProgDist\Progetti\HelloClient
>java HelloClient
Ricevuto:Ciao!

C:\Documents and Settings\vitsca\Desktop\Didattica\ProgDist\Progetti\HelloClient
>■
```

78

L'ESECUZIONE: RISPONDE IL SERVER

79

```

C:\WINDOWS\system32\cmd.exe - java -Djava.security.policy=policyall HelloImpl

C:\Documents and Settings\vitsca\Desktop\Didattica\ProgDist\Progetti\HelloServer
>java -Djava.security.policy=policyall HelloImpl
Effettuo il rebind...
Pronto!
Saluto Pippo
■

```

79

Probabili problemi

80

- File di policy
 - ▣ non scritto
 - attenzione alle estensioni del file, attenzione al nome!
 - ▣ scritto male
 - typos critici
 - ▣ non messo dove dovrebbe essere nella directory dalla quale lanciamo l'applicazione (classe del server)
- Directory corrente di rmiregistry
 - ▣ deve essere quella di dove si trova il file .class di oggetto remoto, (stub) e interfaccia remota
- Se usate due progetti diversi, si genera lo stub, ma si dimentica di copiare nel progetto del client

80

Conclusioni

81

- L'architettura a layer di Java RMI
 - ▣ Il meccanismo di marshalling
- Il processo di creazione
- Il primo esempio: Hello World!
- Conclusioni



Nelle prossime lezioni:

Java Enterprise