Applicazioni Web a tre livelli

Filippo Bosi
Imola Informatica srl
fbosi@imolinfo.it

Abstract: In questa presentazione vengono illustrati, attraverso una serie di esempi di codice, concetti e nozioni pratiche alla base dello sviluppo di applicazioni distribuite con interfaccia Web ed accesso a database. La stessa applicazione di esempio, scritta in Java, viene proposta in architettura Client/Server ed in architettura a tre livelli con interfaccia implementata utilizzando un ORB CORBA (VisiBroker) ed RMI.

Introduzione

- Seminario con un forte taglio pratico
- Tutti gli strumenti utilizzati sono disponibili su Web
  - JBuilder – http://www.inprise.com
    ambiente di sviluppo Java
  - Tomcat – http://jakarta.apache.org
    servlet engine – ambiente runtime per Servlet – open source
  - MySQL – http://www.mysql.com
    server SQL – open source
  - VisiBroker – http://www.inprise.com
    ORB CORBA
  - J2SDK 1.3.1 – http://www.javasoft.com (Windows)
    http://www.blackdown.org (Linux)

Agenda

- Applicazioni Web in Java
  - Servlet e Request-Response HTTP
  - Passaggio di parametri
- Collegamento a database: Client Server
  - Esempio: Login
- Applicazioni a tre livelli
  - Modello Concettuale
  - Implementazione di un'applicazione Web distribuita con application server CORBA
    - Fault Tolerance e Load Balancing (VisiBroker)
  - Implementazione di un'applicazione Web distribuita con application server RMI
  - Utilizzo di Proxy per nascondere il middleware

Servlet

- Paradigma request-response
- E' tipico del protocollo *http*
- Due tipi di richieste fondamentali: GET e POST
- GET – richiesta da parte di un client di lettura di informazioni sul web server
  (es. richiesta di un documento attraverso click su di un link, oppure attraverso un bookmark)
- POST – spedizione da parte di un client di un insieme di dati al web server
  (es. form con i dati anagrafici, oppure username e password per accesso ad un servizio riservato)

Servlet: SimpleGet/1

- SimpleGet
  - Alla ricezione di una richiesta, la servlet genera una pagina html statica

```
(…)
out.println("<html>");
out.println("<head><title>Login</title></head>");
out.println("<body>");
out.println("<p>The servlet has received a GET.
  This is the reply.</p>");
out.println("</body></html>");
(…)
```


SimpleGet/2

- Installazione di una servlet. Viene effettuata attraverso un *deployment descriptor XML* (WEB-INF/web.xml)
- Il deployment descriptor lega la classe Java ad un url del web server

```
<web-app>
  <servlet>
    <servlet-name>simpleget</servlet-name>
    <servlet-class>webapp.SimpleGet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simpleget</servlet-name>
    <url-pattern>/simpleget</url-pattern>
  </servlet-mapping>
</web-app>
```

SimpleGet/3

- Installazione su Tomcat
  - Esecuzione di Tomcat

    *nix: $TOMCAT_HOME/bin/startup.sh
    windows: %TOMCAT_HOME%\bin\startup.bat

  - Creazione di un file *WAR* (Web application Archive). E' un file .jar contenente le classi necessarie al funzionamento delle servlet e il file di descrizione dell'applicazione *web.xml*

$$ *jar -tf webCorbaApp.war*

WEB-INF/classes/webcorbaapp/SimpleGet.class
WEB-INF/web.xml
META-INF/MANIFEST.MF

- Il file *.war* deve essere copiato nella directory *$TOMCAT_HOME/webapps*

SimpleGet/4

- Esecuzione
  - Lancio da un browser dell'URL

http://localhost:<port>/webApp/simpleget

La porta utilizzata da Tomcat viene visualizzata nel log alla partenza

[root@filippo bin]# ./startup.sh
(...)
2001-12-18 12:24:44 - ContextManager: Adding context Ctx(/webApp)
2001-12-18 12:24:46 - PoolTcpConnector: **Starting HttpConnectionHandler on 8080**
2001-12-18 12:24:46 - PoolTcpConnector: Starting Ajp12ConnectionHandler on 8007
(...)

SimplePost/1

- Passaggio di parametri
- Sono necessarie 2 Servlet
  - Una servlet che genera il form di input (**Login**)
  - Una servlet (**SimplePost**) che, rispondendo ad una richiesta di POST, elabora i parametri passati dal form

SimplePost/2

- Form HTML (*Login.java*)

```
out.println("<form method=post action=simplepost>");
out.println("<BR>");
out.println("Immettere identificativo utente e password");
out.println("<BR>");

out.println("<INPUT TYPE=text NAME=\"userid\">");
out.println("<INPUT TYPE=password NAME=\"password\">");

out.println("<br> <br>");
out.println("<INPUT TYPE=submit name=\"submit\"
   value=\"login\">");
out.println("</form>");
```

SimplePost/3

- SimplePost.java – lettura dei parametri

```
Enumeration e=request.getParameterNames();
  while (e.hasMoreElements()) {
    String paramName=(String)e.nextElement();
    out.print("parametro:"+paramName);
    String value =
       (String)request.getParameter(paramName);
    out.println(" valore:"+value);
    out.println("<br>");
```


Client Server

- Interazione della servlet con un server di database
- Utilizzo delle api JDBC (Java DataBase Connectivity)
- La servlet comunica direttamente con il database

- Driver JDBC per mysql:
  http://mmmysql.sourceforge.net/

Client Server/2

- Creazione del database

```
CREATE TABLE UTENTI (
    USERID VARCHAR(10) NOT NULL PRIMARY KEY,
    PASSWORD VARCHAR(10) NOT NULL,
    NOMECOGNOME VARCHAR(20));

INSERT INTO UTENTI(USERID, PASSWORD, NOMECOGNOME)
    VALUES('fbosi', 'filippo', 'Filippo Bosi');
```


Client Server/3

- Verifica della correttezza di un login

La query

```
SELECT NOMECOGNOME FROM UTENTI
WHERE USERID=?
AND PASSWORD=?
```

deve restituire un insieme di righe (ResultSet) non vuoto

Client Server/4

- Esempio di Servlet che accede a database via JDBC (ClientServerPost.java)
- Problemi:

Codice di DB/logica applicativa inserito all'interno del codice di visualizzazione. (Problemi di manutenzione)

Non chiara individuazione delle funzioni offerte dall'applicazione. Manca un'interfaccia di business.

Impossibilità di distribuire l'applicazione su più ambienti di esecuzione (tranne l'ovvia separazione fra DB e Servlet)

Superare il client-server

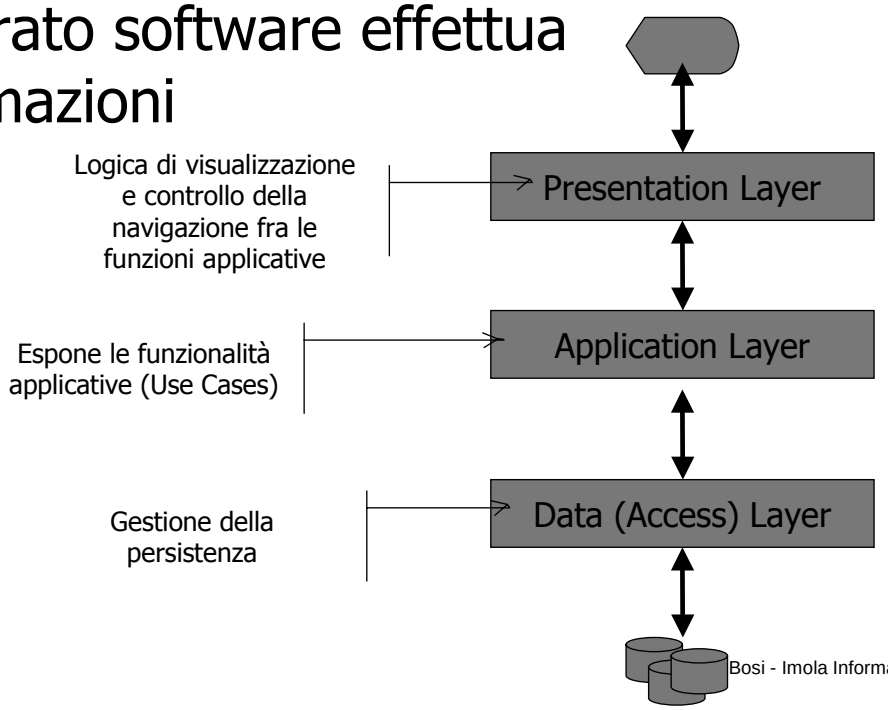
- Architetture a layer (strati)
- Ogni strato individua una particolare responsabilità
- Ogni strato dipende solo dagli strati ad esso contigui
- Il confine fra strati può essere non solo quello di un processo, ma anche una semplice separazione logica all'interno dello stesso processo.

3 Livelli

- E’ un caso notevole di architettura a layer
- Suddivisione di responsabilità su tre livelli: Visualizzazione, Logica Applicativa, Persistenza

3 Livelli: Modello Concettuale

- Ogni strato software effettua trasformazioni



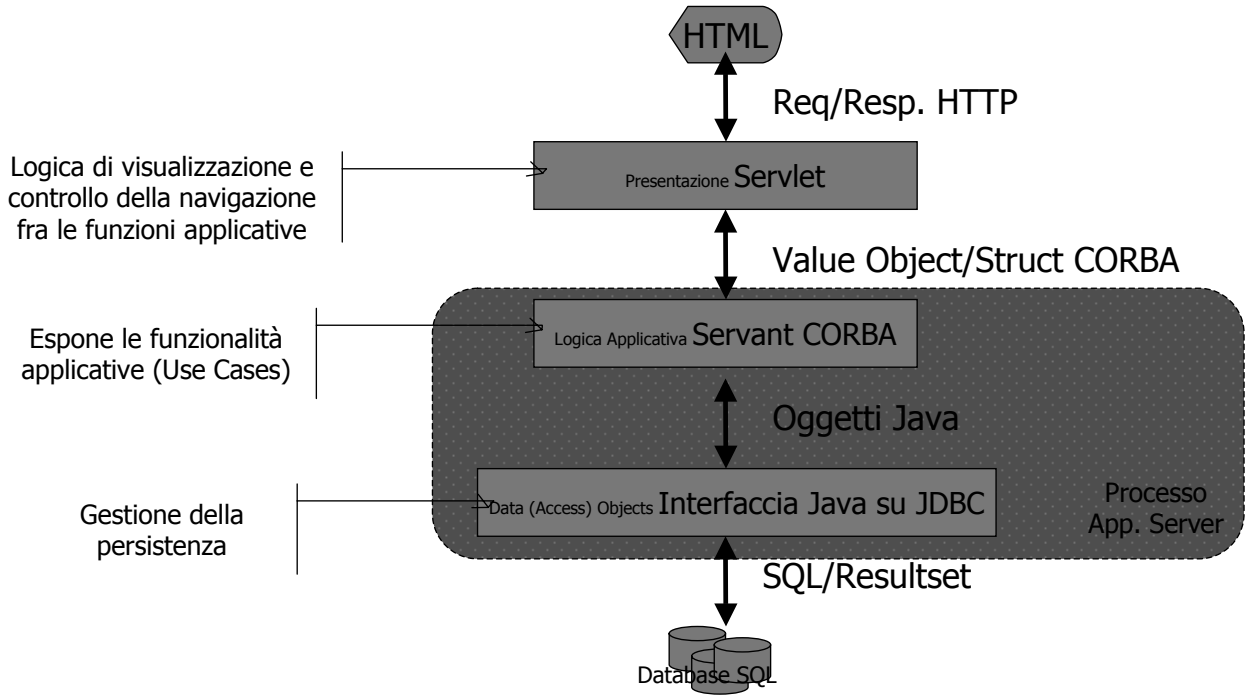
Applicazioni Web/Java

- Presentazione: Servlet/JSP/CGI
- Logica Applicativa: Application Server con interfaccia CORBA oppure RMI.
- Persistenza: JDBC, Object-Relational mapper

Implementazione di un'applicazione Web a servizi con CORBA e Java

- 3 strati software
  - PRESENTAZIONE
  - LOGICA APPLICATIVA
  - PERSISTENZA
- 2 processi
  - SERVLET ENGINE (Tomcat)
  - APPLICATION SERVER (Server CORBA)
- Lo strato di presentazione è realizzato con Servlet
- Lo strato di logica applicativa espone un'interfaccia CORBA
- Lo strato di persistenza interfaccia l'applicazione con il database attraverso JDBC
- Logica applicativa e Persistenza sono contenuti in un unico processo (Application Server), la logica di presentazione è ospitata all'interno di un Servlet Engine/Web Server (Tomcat)

Implementazione



Requisiti

- Controllo di login.

L'utente inserisce username e password, il sistema deve ricercare la coppia su un database e rispondere di conseguenza, restituendo un oggetto di autenticazione, che contiene nome e cognome, oppure visualizzando un messaggio di errore, nel caso in cui non venga trovato lo username oppure la password sia errata.

Modello del problema

- Oggetti di Business

  LoginInfo: UserId, Password

  informazioni di login immesse da un utente
  (identificativo utente e password)

  UserProfile: UserId, CognomeNome

  dati di un utente autenticato
  (identificativo utente e cognome e nome)

- Funzioni di Business (implementazioni di Use Case)

  UserProfile **doLogin**(LoginInfo)

  Effettua un'operazione di *login* passando username e password. Si ottiene in risposta un profilo utente (contenente il nome dell'utente autenticato). Se l'utente non viene trovato, deve essere segnalata una condizione di errore, attraverso il lancio di un'eccezione

Descrizione in IDL CORBA

```
module Login {
    exception UnknownUserPassword {};

    struct UserProfile {
            string userId;
            string cognomeNome;
    };

    struct LoginInfo {
            string userId;
            string password;
    };

    interface LoginManager {
            UserProfile doLogin(in LoginInfo login)
                    raises (UnknownUserPassword);
    };
};
```

Variabili di environment per VisiBroker

- CLASSPATH=$VBJ_HOME/lib/vbjorb.jar
  - Contiene le librerie dell'ORB. Va utilizzata sia per i client, sia per i server VisiBroker
- PATH=$VBJ_HOME/bin
  - Utility a linea di comando
    - *osagent*
      daemon che implementa sistema di directory distribuita per oggetti CORBA VisiBroker. Supporta configurazioni load-balanced e fault-tolerant di servant CORBA. *NON è standard CORBA.*
    - *osfind*
      utility per elencare gli osagent disponibili in rete e tutte le istanze di servant registrati sulla directory distribuita
    - *idl2java*
      E' il traduttore di interfacce descritte in IDL per il linguaggio Java. Crea un insieme di classi *dipendenti dall'ORB* che permettono di realizzare Server e Client CORBA per l'interfaccia che si è compilata

Compilazione dell'IDL

- idl2java login.idl
  - Produce stub e skeleton
  - Stub è un proxy per il client (classe Java che nasconde i dettagli di interfaccia con CORBA e di networking) al client
  - Skeleton: è la base per l'implementazione di un Servant (servente) CORBA.

idl2java

- Oltre a generare Stub e Skeleton, idl2java produce una serie di classi accessorie per interfacciamento con l'ORB, permettendo di scegliere fra diverse implementazioni (ad es. BOA oppure POA)

$ *idl2java –boa login.idl*

- Nel caso del nostro esempio vengono generati:
  - LoginInfo.java, UserProfile.java – struct
  - UnknownUserPassword.java – exception
  - LoginManager – interfaccia

  Contiene la signature Java dei metodi del servant
  - _LoginManagerImplBase – classe di base per l'implementazione del servant

  Inoltre

  Classi xxxHelper (classi che contengono funzioni di utility, come ad esempio per fare cast fra tipi differenti di oggetti CORBA) per ogni interfaccia e struttura

  Classi xxxHolder (pattern per implementare parametri per riferimento in Java)

Rappresentazione Java della struct *LoginInfo*

IDL:

```
struct LoginInfo {
        string userId;
        string password;
   };
```

JAVA:

```
public final class LoginInfo implements
   org.omg.CORBA.portable.IDLEntity {
   public java.lang.String userId;
   public java.lang.String password;

   public LoginInfo(final String userId, final String password) {
   (…)
   }
}
```

Rappresentazione Java dell'interfaccia *LoginManager*

IDL:

```
interface LoginManager {
   UserProfile doLogin(in LoginInfo login) raises
   (UnknownUserPassword);
};
```

JAVA:

```
public interface LoginManager extends
   com.inprise.vbroker.CORBA.Object, Login.LoginManagerOperations,
   org.omg.CORBA.IDLEntity {}

public interface LoginManagerOperations {
   public Login.UserProfile doLogin(Login.LoginInfo login) throws
   Login.UnknownUserPassword;
}
```


Servant

- IDL descrive interfacce, non fornisce implementazioni
- Le implementazioni vengono fornite dagli oggetti Servant, scritti in linguaggi per cui vi sia un mapping IDL
- In VisiBroker, idl2java fornisce una classe di base (Skeleton) di implementazione del Servant. E' sufficiente ereditare da questa e implementare le funzioni dell'interfaccia IDL.

Servant/2

```
public class LoginManagerImpl extends Login._LoginManagerImplBase
   {
  public LoginManagerImpl() {
  }
  public LoginManagerImpl(String name) {
    super(name);
  }

  public Login.UserProfile doLogin(Login.LoginInfo login) throws
   Login.UnknownUserPassword {
    return dataobj.LoginDAO.doLogin(login);
  }

}
```


Realizzazione del server

- Il server implementa il processo che contiene e manda in esecuzione il servant CORBA.
- Il server effettua le seguenti operazioni:

1) Istanzia una copia del servant.

2) Registra il servant in un sistema di naming, per renderlo raggiungibile dai client indipendentemente dalla sua collocazione.

3) Mette in attesa infinita il processo, lasciando il servant in ascolto sul bus CORBA.

Server

- È un'applicazione Java.

```
public static void main(String[] args) {
 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
 com.inprise.vbroker.CORBA.BOA boa =
     ((com.inprise.vbroker.CORBA.ORB)orb).BOA_init();

 Login.LoginManager manager = new
     LoginManagerImpl("LoginServer");

 boa.obj_is_ready(manager);
  System.out.println(manager + " is ready.");

 // lascia aperto il processo in attesa di messaggi da client
  boa.impl_is_ready();
}
```

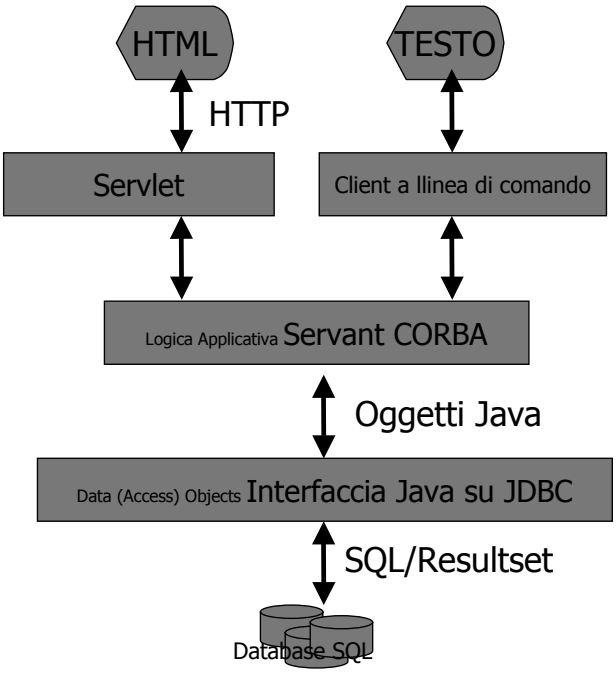

Client

- Ricerca i servant su un sistema di naming (VisiBroker bind())
- Una volta ottenuto un riferimento ad un oggetto ne invoca le funzioni offerte, come se l'oggetto fosse locale.

```
org.omg.CORBA.ORB orb =
    org.omg.CORBA.ORB.init(args,null);
LoginManager manager =
LoginManagerHelper.bind(orb, "LoginServer");
UserProfile profile = manager.doLogin(
    new LoginInfo("fbosi","filippo"));
System.out.println("login effettuato con successo:"
    +profile.toString());
```

Due versioni del client per lo stesso server

- Con applicazioni a layer è estremamente semplice offrire più interfacce per la stessa applicazione.
  - cmdline.Client – client a linea di comando
  - servlet.LoginPost – client web (Servlet)



Processo di bind degli oggetti

*proprietario VisiBroker

osagent contiene un catalogo di associazioni fra IOR (identificativi unici di oggetti Corba) e caratteristiche del servant, come la *classe* del servizio e il *nome* di registrazione, più l'indirizzo IP su cui è attivo il Servant.

Un client richiede l'oggetto per "caratteristiche": generalmente la classe del servizio (collegata all'Helper), che individua una particolare interfaccia.

Se un client non specifica particolari richieste, osagent restituisce uno IOR con un algoritmo round-robin (Load Balancing)

Es. Ior1 e Ior3 sono indistinguibili se non richiedo un particolare IP

IOR	Classe	Nome	IP
Ior1	LoginManager:1.0	LoginServer	x.x.x.x
Ior2	BankManager:1.0	BankServer	x.y.w.z
Ior3	LoginManager:1.0	LoginServer	x.y.y.y

Fault Tolerance

*proprietario VisiBroker

- Utilizzando gli stessi principi di sostituibilità fra oggetti, le librerie VisiBroker sono in grado di effettuare failover a caldo tra oggetti indistinguibili senza nessuna necessità di codifica da parte del client.

- Esempio di failover.
- Presupposto: tutti i servant non devono mantenere uno stato conversazionale con il client (interazione di tipo stateless) oppure devono essere in grado di gestire esplicitamente un failover dello stato

A cosa servono i layer?

👎 Aumento del tempo di scrittura delle applicazioni (stimabile in circa 30%)

👍 Pulizia del codice.
Migliore manutenibilità.
Localizzazione degli errori e degli impatti dei cambiamenti.

👍 Flessibilità:

possibilità di spezzare nei punti di interfaccia le applicazioni su più processi (Scalabilità, tolleranza ai guasti)

possibilità di modificare le applicazioni agendo sulle implementazioni dei layer, senza conseguenze per gli altri layer

Esempio: Cache

- Con una semplice modifica sul codice del servant, si introduce uno schema di caching che permette di evitare letture multiple dal DB degli stessi dati (appserver/CacheServer.java)

```
Hashtable cache=new HashTable();

public Login.UserProfile doLogin(Login.LoginInfo login) throws
   Login.UnknownUserPassword {
    UserProfile
   cachedProfile=(UserProfile)cache.get(login.userId);
    if (cachedProfile==null) {
      System.out.println("cache miss! ["+login.userId+"]");
      UserProfile theProfile=dataobj.LoginDAO.doLogin(login);
      System.out.println("aggiungo in cache...");
      cache.put(login.userId,theProfile);
      return theProfile;
    }

    System.out.println("cache hit! ["+login.userId+"]");
    return cachedProfile;
  }
```


Java e RMI

- CORBA non è l'unico modello di oggetti distribuiti per Java
- RMI: Remote Method Invocation

Fornito insieme al JDK

Gratuito

Inefficiente (implementazione JDK)

Implementazione *solo* Java → più semplice e naturale di Java/CORBA dal punto di vista del programmatore

l'IDL (linguaggio di definizione delle interfacce) è Java stesso (le interfacce remote sono interfacce Java)

CORBA vs RMI

	CORBA	RMI
Naming	osagent (VBJ) Naming Service	rmiregistry
Protocollo	IIOP	JRMP IIOP (RMI/IIOP)
Multi-Linguaggio	SI	NO
Multipiattaforma	SI	SI
Portabilità delle Implementazioni	NO*	SI*
Trasferimento del codice di implementazione degli oggetti	NO	SI
Objects by Value	SI (standard più aggiornati, non supportato da tutti gli orb)	SI
GC Distribuita	NO	SI
Efficienza/Stabilità	SI (prodotto maturo)	NO (impl. JDK)

Implementazione RMI

- Oggetto remoto RMI:

È un oggetto che implementa una *remote interface*

La *remote interface* estende *java.rmi.Remote*

Ogni metodo dell'interfaccia dichiara *java.rmi.RemoteException* nella clausola *throws*

Oggetti Remoti vs Locali

- Quando viene passato un oggetto remoto da una VM ad un'altra, di questo viene passato il riferimento remoto.

Ogni richiamo di funzione, come per i servant CORBA, è un accesso alla rete. A differenza dell'implementazione RMI del JDK, con molti ORB CORBA disponibili in commercio (VisiBroker, Orbix, ecc.) la chiamata di funzione fra oggetti remoti nello stesso spazio di indirizzamento viene ottimizzata, divenendo di fatto una chiamata locale.

- Quando viene passato un oggetto non remoto da una VM ad un'altra, questo è passato per copia attraverso la *serializzazione.*

Per questo motivo tutti gli oggetti locali (bizobj.*) passati come parametri di funzioni di classi remote *devono* essere Serializzabili. La Serializzazione aggiunge notevole overhead nel traffico di rete. Per grafi di oggetti particolarmente complessi può comportare traffico di rete di un ordine di grandezza superiore rispetto ai dati effettivamente trasportati

costruzione

- Definizione dell'interfaccia Remota (*LoginManager*)
- Implementazione degli oggetti Locali (*bizobj.**)
- Implementazione degli oggetti Remoti (*LoginManagerImpl*)
- Implementazione dei Client (*servlet.**)

- Compilazione degli stub e degli skeleton
  - *rmic* – RMI Compiler

  rmic appserver.LoginManagerImpl

  (rmic si esegue sull'implementazione, al contrario di quanto avviene con CORBA)

RMI – Interfaccia

- Le interfacce sono descritte in Java

```
public interface LoginManager extends java.rmi.Remote {
  public UserProfile doLogin(LoginInfo login) throws
  RemoteException;
}
```

- L'implementazione estende UnicastRemoteObject

RMI - Implementazione

- La classe che implementa un servizio deve estendere *UnicastRemoteObject*
- Ogni funzione lancia una *RemoteException*. RemoteException può incapsulare altre eccezioni: questo pattern semplifica il trattamento delle eccezioni nei Client.

```
public UserProfile doLogin(LoginInfo login) throws RemoteException {
   try {
    return dataobj.LoginDAO.doLogin(login);
    } catch (UnknownUserPasswordException e) {
     throw new RemoteException("errore in doLogin",e);
    }
 }
```

Esecuzione e Limitazioni

- Lancio del naming Service

  $ rmiregistry
- Il naming service deve girare sulla stessa macchina dove girano i server
- Il client, al contrario di quanto accade con osagent, deve conoscere la collocazione in rete del server (*implementazione JDK)
- Non viene supportato load balancing, né fault-tolerance. Se si lancia il server più di una volta, viene registrata nel registry solo l'ultima istanza. (*implementazione JDK)

Confronti fra implementazione RMI e CORBA

- Client: effettua sostanzialmente gli stessi passi. Lookup (recupero del riferimento remoto) e richiamo dei metodi.
- Server: implementa un'interfaccia ereditando codice da uno skeleton costruito a partire dall'interfaccia di business (rmic – RMI; idl2java – CORBA)
- In RMI il codice degli oggetti di business deve essere esplicitamente scritto dal programmatore (bizobj.*), mentre in CORBA viene generato a partire dalla descrizione IDL delle strutture dati passate per valore.
- La struttura dell'applicazione è sostanzialmente la stessa

Convergenza fra RMI e CORBA

- Implementazione RMI over IIOP

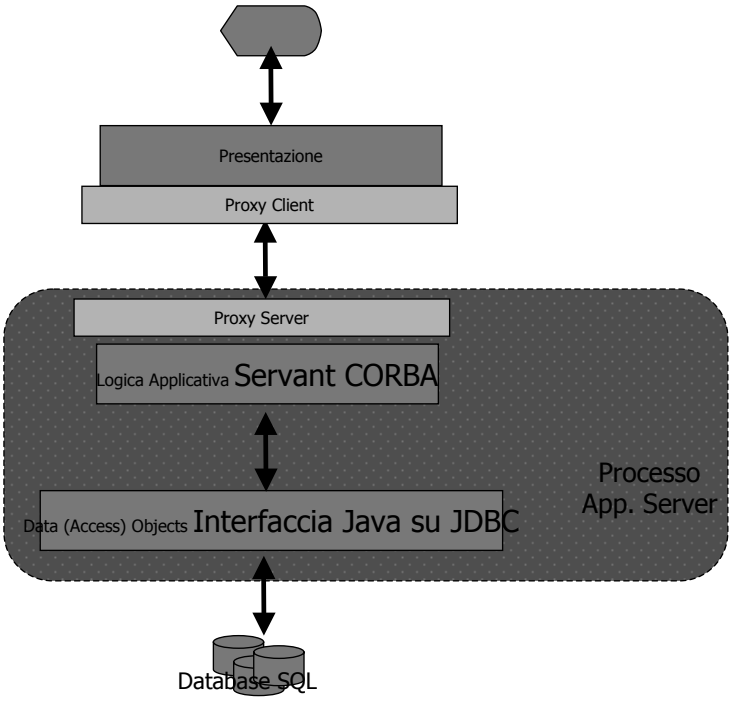
IIOP e le ultime versioni di CORBA supportano il passaggio di oggetti per valore. Questo permette di implementare la modalità RMI di programmazione utilizzando IIOP come protocollo di comunicazione. → Applicazioni Java/RMI in grado di essere interoperabili con altri linguaggi.

RMI over IIOP è considerato talmente importante nello sviluppo Enterprise, da essere stato inserito come uno dei requisiti per la certificazione J2EE per gli Application Server (EJB)

Generalizzazione delle applicazioni a 3 livelli

- Data la sostanziale uniformità delle architetture mostrate, indipendentemente dall'utlizzo di RMI e CORBA, spesso nelle applicazioni Enterprise si tende ad isolare l'applicazione dal middleware attraverso l'utilizzo di proxy.
- Obiettivo: diminuire la dipendenza del codice dalle scelte del middleware

Generalizzazione delle applicazioni a 3 livelli/2



I proxy nascondono i dettagli del middleware. La logica applicativa diventa indipendente, quindi è possibile il riutilizzo al 100% per applicazioni con diversi protocolli di comunicazione, semplicemente sostituendo i Proxy.

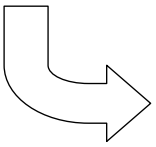
Modifiche nella versione Proxy

- L'interfaccia di business (LoginManager) diventa un'interfaccia normale, e non più un'interfaccia remota (extends Remote)
- L'eccezione lanciata dall'interfaccia LoginManager non è più una RemoteException (rmi) ma una ServiceException (bizobj.*)
- Vengono rimosse tutte le dipendenze dal middleware (import java.rmi.*) nelle parti di business del programma. Queste sono relegate nei soli proxy (Client e Server)
- Vengono create interfacce remote specifiche per i protocolli (es. LoginManagerRMIInterface)
- Creazione di una classe ProxyServer che implementa l'interfaccia di business in senza avere codice dipendente dal middleware (a differenza di LoginManagerImpl)
- Creazione di una classe ProxyClient che separa il client dalle dipendenze dal middleware, nascondendo ad esempio i dettagli di lookup del server
- Aggiunta di una ServiceException, eccezione analoga alla RemoteException di RMI in cui incapsulare tutte le eccezioni create negli strati di business e di middleware, di modo da semplificare notevolmente il trattamento delle eccezioni.

Impatti sul codice di implementazione del Client

- Dalla Servlet (LoginPost) sparisce il codice legato ad RMI

```
try {
    LoginManager manager =
            (LoginManager) Naming.lookup("//filippo/LoginServer");
    profile=manager.doLogin(loginInfo);
  } catch (RemoteException ex) {
    out.println("Eccezione: "+ex.toString());
    profile=null;
  }
```



```
try {
    LoginManager manager = new LoginManagerProxyClient();
    profile=manager.doLogin(loginInfo);
  } catch (ServiceException ex) {
    out.println("Eccezione: "+ex.toString());
    profile=null;
  }
```

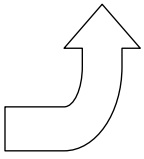

Impatti sul codice di implementazione del Server

- Dall'implementazione del processo server sparisce il codice legato ad RMI

```
  public static void main(String[] args) {
    // Inizializza il security manager
    if (System.getSecurityManager()==null)
      System.setSecurityManager(new
RMISecurityManager());

    // Registrazione su rmiregistry
    String serverName="//filippo/LoginServer";
    try {
      // Istanzio il Server
      LoginManager manager=new LoginManagerImpl();
      System.out.println("sto registrando
"+serverName+"...");
      // registro nel naming service (rmiregistry)
      Naming.rebind(serverName, manager);
      System.out.println("LoginManagerImpl
registrato");
      } catch (Exception e) {
        System.err.println("eccezione:
"+e.getMessage());
        e.printStackTrace();
      }
  }
```

```
public static void main()
{
    new LoginManagerProxyServer();
}
```



Vantaggi della versione Proxy

- Indipendenza del codice di business dal middleware utilizzato
  - Possibilità di cambiare il middleware con impatto sui soli Proxy
  - Maggiore pulizia del codice (le parti business hanno meno codice, e comunque codice non legato al middleware)
  - Possibilità di utilizzare programmatori che non hanno competenze di utilizzo delle classi middleware per lo sviluppo di parti di codice business (risparmio $$$)
- Svantaggi: maggior numero di classi e di file
  - N.B. Questo non è uno svantaggio, soprattutto se si parla con un architetto che ama la programmazione OO in Java!

Struttura della directory dei sorgenti di esempio

- /webApp – esempi di Servlet
- /clientServerWebApp – versione client/server
- /corbaWebApp – versione CORBA VisiBroker
- /rmiWebApp – versione RMI
- /proxyWebApp – versione RMI con Proxy
- /sql – script sql per la creazione del database degli esempi (MySQL)

- Script all'interno di ogni directory di esempio
- ./startTomcat.sh ./stopTomcat.sh – script di avvio e spegnimento di Tomcat
- ./startAppServer.sh – script di avvio del processo Server
- ./startRMIregistry.sh – script di avvio dell'RMIRegistry (impl. RMI)
- ./startOsAgent.sh – script di avvio di OSAgent (impl. VisiBroker)
- ./deploy.sh – installazione del .war su Tomcat (da effettuare a Tomcat spento)

DOMANDE?