



Java Management Extensions (JMX)

Università di Bologna
CdS Laurea Magistrale in Ingegneria Informatica
I Ciclo - A.A. 2013/2014

Corso di Sistemi Distribuiti M

Monitoraggio e Gestione tramite JMX

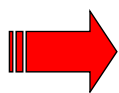
Docente: Paolo Bellavista
paolo.bellavista@unibo.it

<http://lia.deis.unibo.it/Courses/sd1314-info/>
<http://lia.deis.unibo.it/Staff/PaoloBellavista/>



Controllo e Monitoraggio Distribuito di Componenti

Come già detto più volte, siamo interessati non solo alla fase di sviluppo, ma ***principalmente alla fase di deployment, configurazione ed esecuzione in ambiente reale***



Necessità di controllo on-line e conseguenti azioni di gestione (non solo di network equipment ma anche di componenti applicativi e di servizio)

Obiettivi: fault detection, misura di performance e riconfigurazione/re-deployment, identificazione colli di bottiglia, ...

Con quali pro e quali contro?

“Soliti” modelli: ***push*** vs. ***pull***, ***reattivi*** vs. ***proattivi***, ***ottimistici*** vs. ***pessimistici***, con manager ***centralizzato*** vs. ***parzialmente distribuito*** vs. ***completamente distribuito***, ...



Controllo e Monitoraggio Distribuito di Componenti

Importanza di standard per controllo distribuito in **ambienti aperti e interoperabili** (la stessa necessità di standard, a che cosa ha portato nel mondo del networking?)

Solo per fare un esempio, Distributed Management Task Force (DMTF) è un'organizzazione per la **standardizzazione per IT system management** in ambienti industriali e Internet. Gli standard DMTF permettono la costruzione di componenti per system management **indipendenti dalla piattaforma e technology-neutral**, abilitando così **interoperabilità** fra prodotti per la gestione di sistemi di **diversi vendor**

Alcuni elementi fondamentali in DMTF:

- ❑ **Common Information Model (CIM)**
- ❑ **Common Diagnostic Model (CDM)**
- ❑ **Web-Based Enterprise Management (WBEM)**



Controllo e Monitoraggio Distribuito di Componenti

- ❑ **Common Information Model (CIM)** – modello astratto per la **rappresentazione degli elementi gestiti** (ad esempio, computer o storage area network) come **insieme di oggetti e relazioni**. CIM è **estensibile** per consentire l'introduzione di estensioni product-specific

Qualcuno si ricorda qualcosa di **SNMP e CMIB**?

- ❑ **Common Diagnostic Model (CDM)** – **modello di diagnostica** e definizione di come questo debba essere incorporato nell'infrastruttura di management

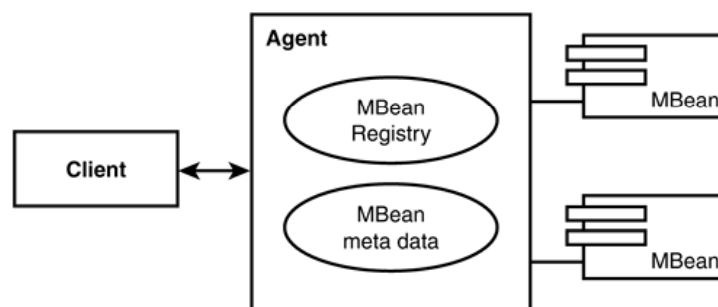
- ❑ **Web-Based Enterprise Management (WBEM)** – **protocolli** per l'interazione fra componenti di system management (conformi a CIM e ai suoi profili) e la loro interrogazione

Qualcuno ha mai visto altrove il concetto di **profilo**?



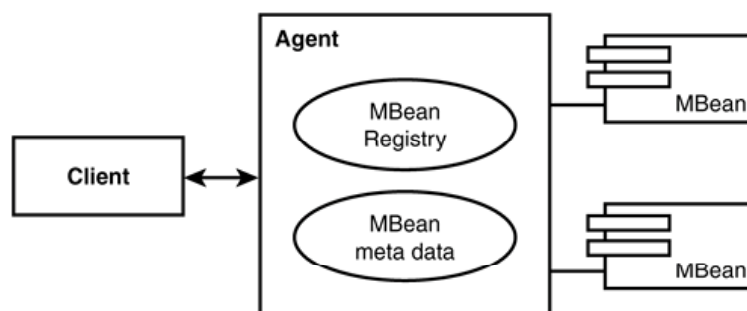
Panoramica Generale

- ❑ Prima di Java Management Extensions (JMX) non vi era **nessun approccio standardizzato** in Java per far **partire, gestire, monitorare e fermare** l'esecuzione di componenti software
- ❑ Componenti software conformi alla specifica JMX vengono chiamati **MBean** (Managed Bean)
- ❑ Componenti MBean sono gestiti attraverso un **agente**:
 - che svolge il ruolo di **registry**
 - che offre alle applicazioni di management (*clienti di tale agente*) un modo di **effettuare query e modificare i bean** gestiti



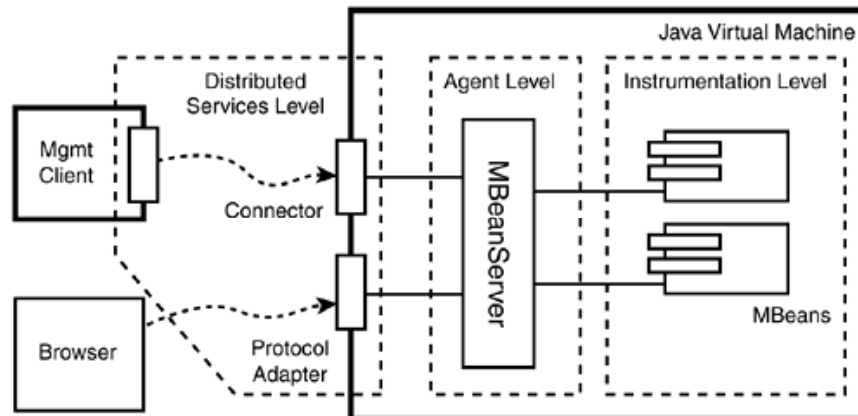
Panoramica Generale

- ❑ JMX realizza e sfrutta, ancora una volta, il ben noto **principio di decoupling**. Tutte le comunicazioni fra clienti e MBean avvengono **attraverso il livello intermedio dell'agente**:
 - i clienti mandano **query** all'agente relative agli **MBean registrati**
 - i clienti chiedono all'agente l'esecuzione di **metodi di business/management** (specificati nell'interfaccia di management) sugli MBean desiderati
- ❑ **NON viene passato MAI alcun riferimento diretto a componenti MBean** (visibilità solo all'interno dell'agente)



JMX è organizzato secondo **un'architettura a tre livelli**:

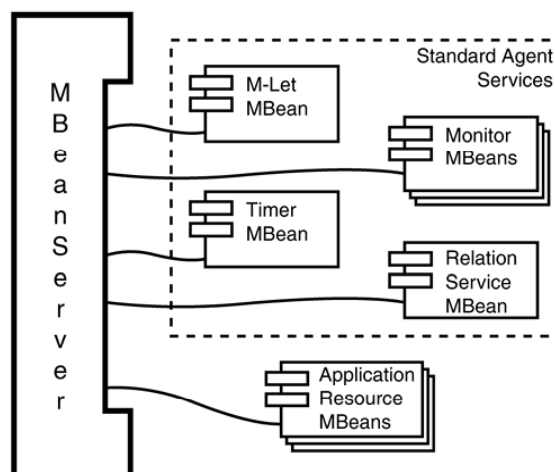
- ❑ I componenti gestiti appartengono al **livello instrumentation**
- ❑ Il livello **agente** è costituito dal **registro per gli MBean** (MBeanServer) e da alcuni servizi standard aggiuntivi
- ❑ Il livello dei **servizi distribuiti** è costituito da **adattatori e connettori** (adapter e connector), necessari per supportare l'accesso remoto al livello agente



- ❑ Livello instrumentation definisce **come creare risorse gestibili tramite JMX (MBeans)**, ovvero **oggetti che offrono metodi** per:
 - gestire un'applicazione
 - gestire un componente software
 - gestire un servizio
 - gestire un dispositivo
 - ...
- ❑ MBean non è altro che un componente che **implementa una interfaccia di gestione, staticamente o dinamicamente**:
 - Nel primo caso, implementa **un'interfaccia Java standard** e l'agente ne fa **inspection** tramite tecniche di **reflection** e **convenzioni sui nomi**
 - Nel secondo caso, offre un insieme di **oggetti metadata** attraverso i quali l'agente riesce a **scoprire i metodi di management esposti**

- ❑ JMX definisce 4 tipi di componenti MBean:
 - **Standard MBean**: creato dichiarando esplicitamente una **interfaccia Java** con **l'informazione di management** che l'oggetto gestito implementa
 - **Dynamic MBean**: un oggetto che implementa l'interfaccia **DynamicMBean** e offre la descrizione dei suoi veri metodi di management attraverso un **insieme di oggetti metadata** che tale interfaccia richiede di fornire
 - **Model MBean**: un DynamicMBean **esteso con descrittori aggiuntivi** che definiscono proprietà aggiuntive (*behavioral properties*, come funzionalità orizzontali di persistenza, sicurezza, ...)
 - **Open MBean** (*non di implementazione obbligatoria per essere conformi alla specifica*): un MBean in cui i **tipi utilizzati** nei metodi di management hanno il vincolo ulteriore di essere **inclusi in un set predefinito** di classi e tipi di base
- ❑ Il supporto ai primi tre tipi di MBean è *mandatory* a partire dalla specifica JMX1.0 (ricordiamo che JMX è una specifica, con diverse implementazioni possibili, come quella di Sun o IBM Tivoli)

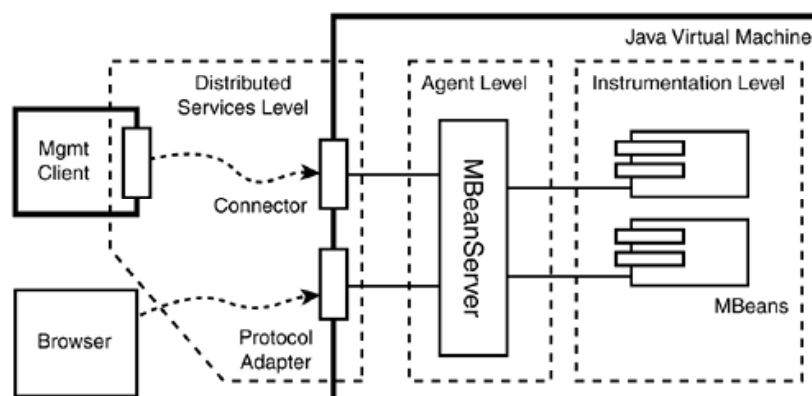
- ❑ Il livello di agente è costituito da un **server MBean** e da un **insieme di servizi di agente** basati sul livello di instrumentation
- ❑ 4 servizi di agente sono definiti nella specifica JMX: **M-Let, Timer, Monitoring e Relation**
- ❑ Il livello agente introduce il concetto di **naming per gli oggetti**, nomi che il lato cliente può utilizzare come **riferimento indiretto alle risorse gestite**



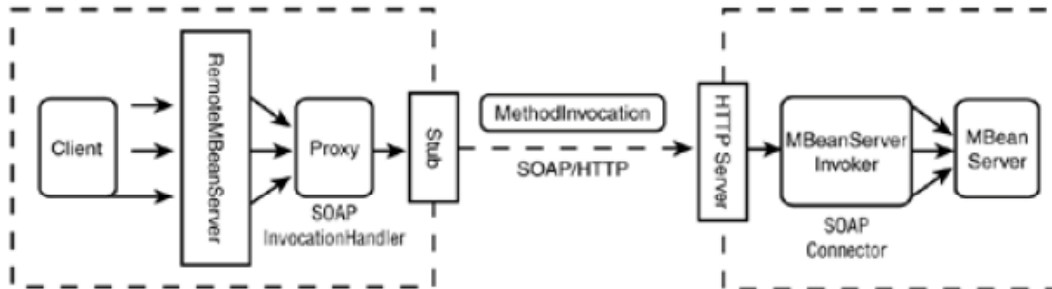
- ❑ Il **server MBean** è uno dei componenti chiave dell'architettura di management:
 - Opera come un **canale di comunicazione** che smista/delega tutte le invocazioni fra applicazioni di management e risorse gestite
 - Espone metodi per la **creazione/effettuazione di query**, per **invocare operazioni** e per **manipolare attributi** su MBean
- ❑ Il tipo di implementazione dei componenti MBean (**Standard, Dynamic, Model, ...**) è **totalmente trasparente** alle applicazioni client-side di gestione

MBeanServer è un oggetto locale alla JVM dei **componenti gestiti** e non offre particolare supporto alla connessione remota verso di sé:

- Servono **connettori JMX o adattatori di protocollo** per accettare chiamate provenienti dall'esterno della JVM
- Questi componenti (connettori/adattatori) sono spesso essi stessi degli **MBean**, registrati sull'agente, e forniscono una pluralità di differenti forme di connettività



- ❑ I **connettori JMX** sono strutturati in **due componenti**:
 - Lato server, l'agente registra un **server per le connessioni** capace di ricevere invocazioni remote di metodo
 - Lato cliente, si può utilizzare una **vista remota del server MBean** per invocare operazioni su di esso



- ❑ **Adattatori di protocollo** (implementazione solo lato server Mbean) possono adattare operazioni server MBean a rappresentaz. secondo determinato protocollo (o anche verso diverso modello di informazioni, come SNMP Management Information Base), permettendo ad applicazioni management legacy o a strumenti non-Java di interoperare con JMX

- ❑ La specifica **JMX Remote API** definisce come si possa fare **l'advertising e trovare agenti JMX** usando **infrastrutture di discovery e lookup esistenti**:
 - La specifica NON definisce un ulteriore servizio di discovery e lookup
- ❑ La tecnologia JMX offre una **soluzione standard per l'esportazione delle API di JMX instrumentation** verso applicazioni remote, **basata su RMI**
- ❑ Inoltre, JMX Remote API definisce un **protocollo opzionale** (non-mandatory e più efficiente) **basato direttamente su socket TCP**, chiamato JMX Messaging Protocol (JMXMP)

- ❑ MBean sono stati progettati per essere **flessibili, semplici e facili da implementare**
- ❑ Gli sviluppatori di applicazioni, servizi di supporto e dispositivi possono rendere i loro prodotti gestibili (*manageable*) **in modo standard**, senza necessità di conoscere a fondo e di investire in sistemi complessi di management
- ❑ Gli **oggetti esistenti possono facilmente essere estesi** per produrre **MBean standard** o essere oggetto di **wrapping come MBean dinamici**, rendendo così le risorse esistenti facilmente manageable a basso costo

Scendiamo in qualche Dettaglio: Standard MBean

Standard MBean

- ❑ Il modo più **semplice** per rendere JMX-managed **nuove classi Java**
- ❑ **Interfaccia statically-typed** dichiara esplicitamente gli attributi (tramite metodi getter e setter) e le operazioni di gestione
- ❑ **Convenzione sui nomi:** quando un managed object viene registrato, l'agente cerca una **interfaccia di management con lo stesso nome dell'oggetto + suffisso MBean** (nel caso, navigando l'albero di ereditarietà della classe)

```
public interface UserMBean{
    public long getId();
    public void setId(long id);
    public boolean isActive();
    public void setActive(boolean active);
    public String printInfo();
}

public class User
    implements UserMBean { ... }

public class Student
    extends User {
    /* anche questa classe può
       essere registrata come un
       UserMBean */
    ... }
```


- ❑ Per **registrare** un manageable object come un MBean è necessario creare prima **ObjectName**
- ❑ Il **riferimento all'agente** può essere ottenuto da una lista di implementazioni disponibili di MBeanServer o creandolo da zero
- ❑ **La registrazione di MBean consiste semplicemente nell'associare il manageable object con il suo nome di oggetto**

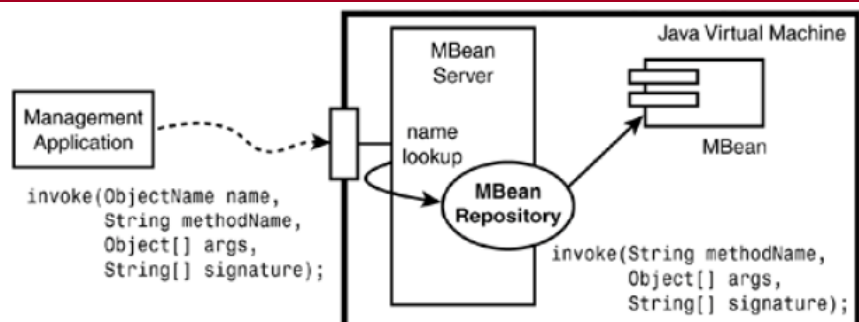
```
ObjectName username =
    new ObjectName(
        "example:name=user1");

List serverList =
    MBeanServerFactory.
        findMBeanServer(null);
MBeanServer server =
    (MBeanServer) serverList.
        iterator().next();

/* oppure per la creazione...
MBeanServer server =
    MBeanServerFactory.
        createMBeanServer(); */

server.registerMBean( new
    User(), username);
```

- ❑ L'applicazione di management **riferisce MBean passando un riferimento a object name** all'agente, **per ogni operazione invocata**
- ❑ Server MBean cerca il riferimento Java corrispondente a MBean nel suo **repository interno** e invoca l'operazione corrispondente (o la modifica dell'attributo) su MBean



```
ObjectName username =
    new ObjectName(
        "example:name=user1");

Object result =
    server.invoke(
        username, // nome MBean
        "printInfo" // nome operaz
        null, // no param
        null); // void signature
```

L'architettura JMX definisce un **meccanismo di notifica** per MBean che consente di inviare eventi verso altri MBean o applicazioni di management

- ❑ **MBean che vogliono emettere eventi** di management devono implementare l'interfaccia **NotificationBroadcaster**
- ❑ **Oggetti listener** per gli eventi devono invece implementare l'interfaccia **NotificationListener** e devono effettuare loro **subscription presso Mbean** (locale o remoto) **che fa da broadcaster**
- ❑ Questa **subscription** è fatta attraverso il **livello di agente**
- ❑ **Operazioni di notifica svolte da broadcaster MBean sono parte della loro interfaccia di management JMX:**
 - Applicazioni possono effettuare **query sul livello agent** per avere info su **quali tipi di notifica** gli MBean di interesse possono emettere
 - A tal fine, MBean broadcaster forniscono oggetti **MBeanNotificationInfo**

- ❑ La classe JMX **Notification** estende **EventObject** introducendo campi per il **tipo di evento, numero di sequenza, timestamp, messaggio** e dati utente opzionali
- ❑ **Le notifiche possono essere filtrate:**
 - Implementazione dell'interfaccia **NotificationFilter** è **subscribed presso broadcaster MBean**, insieme con il listener
 - il broadcaster deve controllare se la notifica supera il filtro **prima di inviarla**

```
public interface
    NotificationFilter {
        public boolean
            isNotificationEnabled(
                Notification notification);
    }

public interface
    NotificationBroadcaster {

        /* ... */

        public void
            addNotificationListener(
                NotificationListener listener,
                NotificationFilter filter,
                Object handback) throws
                    IllegalArgumentException;
    }
```

- ❑ Poiché l'implementazione di broadcaster MBean può diventare anche piuttosto complessa, è **messa a disposizione una classe NotificationBroadcasterSupport** che **implementa l'interfaccia NotificationBroadcaster**. I propri broadcaster MBean possono:
 - O **estendere tale classe per ereditare** quella implementazione dei metodi di broadcasting
 - O **delegare a questa classe il supporto** alla gestione delle registrazioni e all'invocazione delle notifiche
- ❑ Il meccanismo di notifica è generico e adatto a qualsiasi tipo di notifica user-defined. Comunque **JMX definisce la specifica classe AttributeChangeNotification** per MBean che vogliono inviare **notifiche sul cambiamento dei loro attributi di management**

- ❑ **MBean dinamici implementano l'interfaccia generica DynamicMBean** che offre metodi all'agente per fare il **discovery di metodi e attributi di management** (reale interfaccia di gestione)
 - ❑ I metadati che descrivono l'interfaccia di management sono completamente sotto la responsabilità dello sviluppatore
- In Standard MBean i metadati vengono generati dall'agente stesso tramite introspezione
- ❑ Casi possibili di utilizzo di Dynamic MBean:
 - Situazioni in cui **l'interfaccia di management può cambiare spesso**
 - Abilitare **management su risorse esistenti**

```
public interface
DynamicMBean {

    public MBeanInfo getMBeanInfo();

    public Object getAttribute(
        String attribute) throws ... ;

    public AttributeList getAttributes(
        String[] attributes) throws ... ;

    public void setAttribute(
        Attribute attribute) throws ... ;

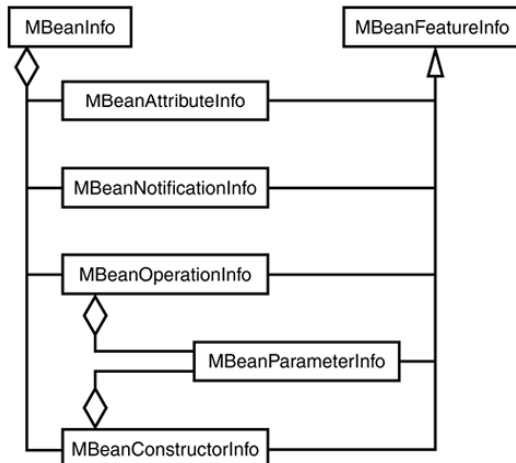
    public AttributeList setAttributes(
        AttributeList attributes) throws ... ;

    public Object invoke(
        String actionName,
        Object[] params,
        String[] signature) throws ... ;

}
```

In MBean dinamici:

- **Interfaccia di management esposta tramite le classi di metadata** definite in JMX API
- **metadata ritrovati dinamicamente** dall'agente come istanza della classe **MBeanInfo**
- ❑ **MBeanInfo** include tutti gli elementi di metadata (che ereditano caratteristiche comuni dalla classe **MBeanFeatureInfo**)



Metodi della classe **MBeanInfo**

<code>public String getClassName()</code>	Restituisce il nome della classe di MBean
<code>public String getDescription()</code>	Restituisce una descrizione di MBean
<code>public MBeanAttributeInfo[] getAttributes()</code>	Restituisce un array di oggetti, uno per ogni attributo di management
<code>public MBeanOperationInfo[] getOperations()</code>	Restituisce un array di oggetti, uno per ogni operazione di management
<code>public MBeanConstructorInfo[] getConstructors()</code>	Restituisce un array di oggetti, uno per ogni costruttore pubblico di MBean
<code>public MBeanNotificationInfo[] getNotifications()</code>	Restituisce un array di oggetti, uno per ogni tipo di notifica che MBean può emettere

```

public class DynamicUser extends NotificationBroadcasterSupport
    implements DynamicMbean {

    // Attributi
    final static String ID = "id";
    private long id = System.currentTimeMillis();
    public Object getAttribute(String attribute) throws
        AttributeNotFoundException, MBeanException, ReflectionException {
        if (attribute.equals(ID)) return new Long(id);
        throw new AttributeNotFoundException("Missing attribute " +
            attribute);
    }

    // Operazioni
    final static String PRINT = "printInfo";
    public String printInfo() { return "Sono un MBean dinamico"; }
    public Object invoke(String actionName, Object[] params, String[]
        signature) throws ... {
        if (actionName.equals(PRINT)) return printInfo(); )
        throw new UnsupportedOperationException("Unknown operation " +
            actionName);
    }
    ...
  
```

```
public MBeanInfo getMBeanInfo() {
    final boolean READABLE = true; final boolean WRITABLE = true;
    final boolean IS_GETTERFORM = true;
    String classname = getClass().getName();
    String description = "Sono un MBean dinamico";

    MBeanAttributeInfo id = new MBeanAttributeInfo(ID, long.class.
        getName(), "id", READABLE, !WRITABLE, !IS_GETTERFORM);
    MBeanConstructorInfo defcon = new MBeanConstructorInfo(
        "Default", "Creates", null);
    MBeanOperationInfo print = new MBeanOperationInfo(PRINT,
        "Prints info", null, String.class.getName(), MBeanOperation-
        Info.INFO);

    return new MBeanInfo(classname, description,
        new MBeanAttributeInfo[] { id },
        new MBeanConstructorInfo[] { defcon },
        new MBeanOperationInfo[] { print },
        null);
}
```

- ❑ **ModelMBean sono estensioni di MBean dinamici**
- ❑ Forniscono un template generico per:
 - Creare un'implementazione di **gestione per risorse esistenti**
 - Separare l'implementazione di management dall'implement. della risorsa
 - **Estendere metadata di gestione** per fornire informazioni aggiuntive e proprietà behavioral:
 - Proprietà di **caching**
 - Proprietà di **sicurezza**
 - Proprietà di **transazionalità**
 - Proprietà di **persistenza**
 - ...
- ❑ Tutti le implementazioni di JMX MBean server devono fornire **almeno una implementazione** dell'interfaccia **ModelMBean** tramite la classe **RequiredModelMBean**

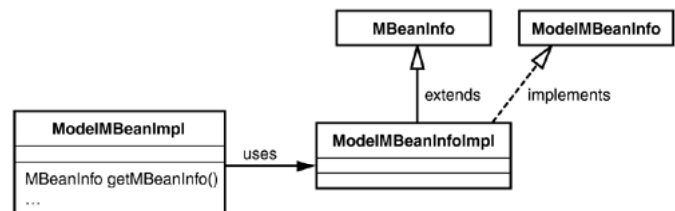
```
public interface ModelMBean
    extends DynamicMBean,
        PersistentMBean,
        ModelMBeanNotificationBroadcaster {
    public void setModelMBeanInfo(
        ModelMBeanInfo inModelMBeanInfo)
        throws ... ;

    public void setManagedResource(
        Object mr, String mr_type) throws ...
    ;
}

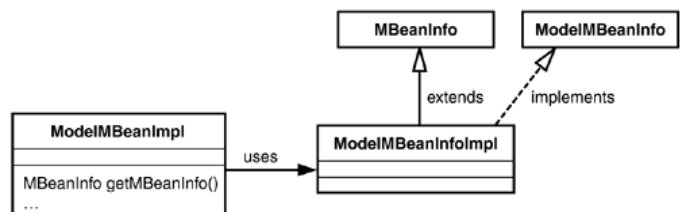
public class RequiredModelMBean
    implements ModelMBean, ... {
    ...
}
```


- ❑ Oggetti che implementano l'interfaccia **Descriptor** sono usati nei metadati di Model MBean per **aggiungere politiche**:
 - Politiche specifiche per la particolare implementazione dell'agente JMX
 - La specifica JMX definisce alcuni comportamenti standard
 - Implementazioni di Model MBean possono essere estese per supportare comportamenti custom
- ❑ **Un descrittore è una collezione di coppie nome-valore in base alle quali l'implementazione dell'agente adatta il suo comportamento**
- ❑ Le classi di metadata di Model MBean estendono le classi corrispondenti usate con MBean dinamici e standard (implicitamente) e implementano l'interfaccia DescriptorAccess

```
public interface Descriptor
    extends Serializable, Cloneable
{
    public String[] getFields();
    public void setField(String
        name, Object value);
    public void removeField(String
        name);
    ...
}
```



- ❑ Oggetti che implementano l'interfaccia **Descriptor** sono usati nei metadati di Model MBean per **aggiungere politiche**:
 - Politiche specifiche per la particolare implementazione dell'agente JMX
 - La specifica JMX definisce alcuni comportamenti standard
 - Implementazioni di Model MBean possono essere estese per supportare comportamenti custom
- ❑ **Un descrittore è una collezione di coppie nome-valore in base alle quali l'implementazione dell'agente adatta il suo comportamento**
- ❑ Le classi di metadata di Model MBean estendono le classi corrispondenti usate con MBean dinamici e standard (implicitamente) e implementano l'interfaccia DescriptorAccess



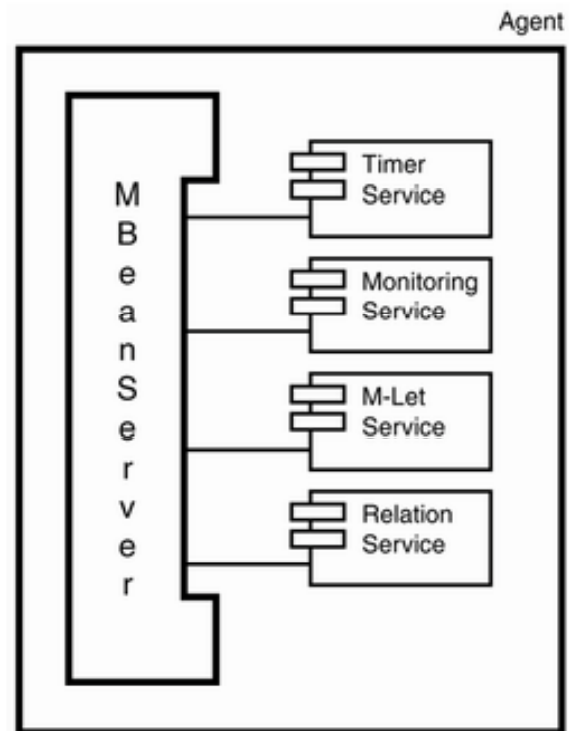
```
public interface DescriptorAccess
{
    public Descriptor getDescriptor();
    public void setDescriptor(
        Descriptor desc);
}

public class ModelMBeanAttributeInfo
    extends MBeanAttributeInfo
    implements DescriptorAccess, Cloneable
{
    ...
}

public class ModelMBeanOperationInfo ...
public class ModelMBeanConstructorInfo
    ...
public class ModelMBeanNotificationInfo
    ...
```


La specifica JMX **definisce 4 servizi** distinti a livello di agente che devono essere **disponibili su ogni implementazione conforme** alla specifica:

- **M-Let Service**: permette agli MBean di essere **caricati dalla rete e inclusi nel livello di agente a runtime**
- **Timer Service**: **scheduler che si occupa dell'invio di notifiche** agli altri MBean
- **Monitoring Service**: MBean che svolge il ruolo di **osservatore per gli attributi di management** degli altri bean e che **notifica le modifiche avvenute**
- **Relation Service**: permette di creare **associazioni fra MBean** e mantiene la loro consistenza



- ❑ **Loading dinamico di nuove classi** Java dal server MBean:
 - Su macchina locale
 - Da macchina remota
- ❑ **Spostamento della configurazione di una applicazione** verso un server remoto
- ❑ Come ogni altro standard MBean, l'interfaccia **MLetMBean** **espone le operazioni di management** considerate rilevanti per il servizio:
 - `addURL()` ;
 - `getMBeansFromURL()` ;
 - ...
- ❑ All'URL specificato si trovano i **file di testo** M-Let che descrivono i componenti MBean tramite **MLET tag**

```
<MLET CODE = class | OBJECT =
    serfile
    ARCHIVE = "archiveList"
    [CODEBASE = codebaseURL]
    [NAME = MBeanName]
    [VERSION = version] >
    [arglist]
</MLET>
```

Ad esempio:

```
<MLET CODE=com.mycompany.Foo

    ARCHIVE="MyComponents.jar,acme.jar"
</MLET>
```

- ❑ Il servizio di **Timer** è basato sul **meccanismo di notifica di JMX**:
 - **TimerMBean** è un broadcaster MBean
 - **Per ricevere notifiche dal timer, il consumatore deve implementare l'interfaccia *NotificationListener* e registrarsi**
- ❑ Analogo al servizio di **cron** in Unix/Linux o a **Task Scheduler Service** su Windows NT

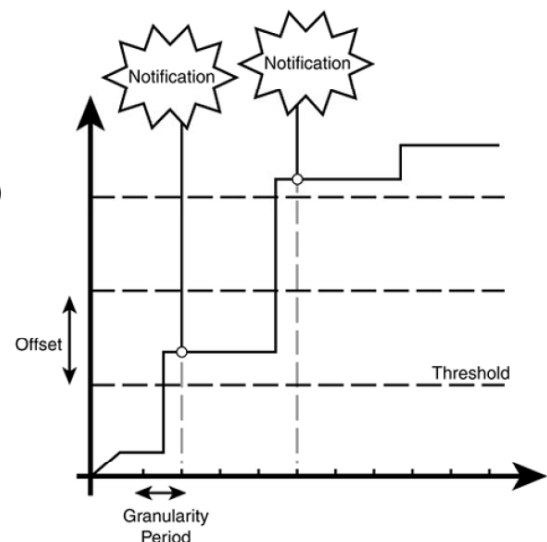
```
// fa partire il servizio di timer
List list =
    MBeanServerFactory.findMBeanServer (null);
MBeanServer server = (MBeanServer)list.
    iterator().next();
ObjectName timer = new ObjectName("service:
    name=timer");
server.registerMBean(new Timer(),timer);
server.invoke(timer,"start",null,null);

// configurazione di notification time
Date date = new Date(System.
    currentTimeMillis()+Timer.ONE_SECOND*5);
server.invoke(timer,           // MBean
    "addNotification",         // metodo
    new Object[] {             // args
        "timer.notification",  // tipo
        "Schedule notification", // messaggio
        null,                  // user data
        date},                 // time
    new String[] { String.class.getName(),
        String.class.getName(),
        Object.class.getName(), // signature
        Date.class.getName() } );

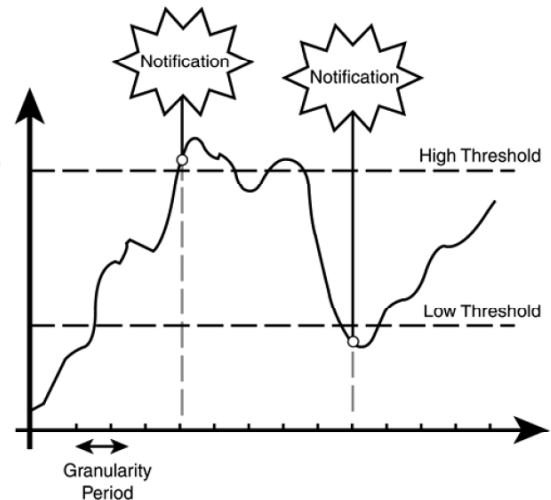
// registra il listener MBean
server.addNotificationListener (timer,this,null,
    null);
```

Sistemi Distrib

- ❑ Un insieme di MBean che possono essere **utilizzati per effettuare il monitoring degli attributi** di risorse gestite
- ❑ Le notifiche dei monitor differiscono dalle usuali notifiche di modifica di attributi perché si possono introdurre **threshold e periodi di granularità**
- ❑ 3 implementazioni differenti:
 - **Counter monitor** traccia le variazioni di attributi che si comportano come contatori (senza variazioni «discontinue»)
 - **Gauge monitor** per attributi integer e float, ad intervalli di granularità configurabile, con threshold
 - **String monitor** per informare in relazione a string matching/dis-matching rispetto a valori attesi



- ❑ Un insieme di MBean che possono essere **utilizzati per effettuare il monitoring degli attributi** di risorse gestite
- ❑ Le notifiche dei monitor differiscono dalle usuali notifiche di modifica di attributi perché si possono introdurre **threshold e periodi di granularità**
- ❑ 3 implementazioni differenti:
 - **Counter monitor** traccia le variazioni di attributi che si comportano come contatori (senza variazioni «discontinue»)
 - **Gauge monitor** per attributi integer e float, ad intervalli di granularità configurabile, con threshold
 - **String monitor** per informare in relazione a string matching/dis-matching rispetto a valori attesi



Servizi Agent-level Standard: Relation

Permette di **definire relazioni fra MBean e di reagire a modifiche** (caso classico: dipendenze)

- Consistenza delle relazioni mantenuta tramite la **definizione di ruoli per gli MBean e associando/disassociando oggetti MBean a ruoli differenti** nelle relazioni
- **Notifiche emesse alla modifica nelle istanze di relazione** (creazione, aggiornamento, rimozione, ...)

```
RoleInfo monInfo = new RoleInfo(
    "Monitor", "javax.management.
    monitor.GaugeMonitor", true, true, 0,
    ROLE_CARDINALITY_INFINITY, // [0,*]
    "Descrizione del monitor");

RoleInfo obsInfo = new RoleInfo(
    "Observable", "examples.
    ThreadMonitor", true, true, 1, 1, // [1,1]
    "Descrizione del ruolo observable");

/* Oggetti relazione implementano
   l'interfaccia RelationType */
RelationTypeSupport relationType =
    new RelationTypeSupport(
        "ObservedMBean", new RoleInfo[] {
            observableInfo, monitorInfo
        }
    );

/* ... */
```

Per **effettuare operazioni remote su MBean**, un **server per connettori RMI** è a disposizione lato server:

- Tramite chiamata alla classe **JMXServiceURL** si crea un nuovo URL di servizio (**indirizzo per il server di connector**)
- Il server di connector RMI è creato via **JMXConnectorServerFactory**, con parametri URL di servizio e MBeanServer
- Il server di connector deve essere messo in **esecuzione**

On the server side:

```
MBeanServer mbs = MBeanServer-
Factory.createMBeanServer();
JMXServiceURL url = new
JMXServiceURL("service:jmx:
rmi:///jndi/rmi://" +
"localhost: 9999/server");
JMXConnectorServer cs =
JMXConnectorServerFactory.
newJMXConnectorServer(url,
null, mbs);
cs.start();
```

URL (in formato JNDI) indica dove reperire uno stub RMI per il connettore (tipicamente in un direttorio riconosciuto da JNDI come RMI registry o LDAP):

- connettore usa il trasporto di default RMI
- registry RMI in cui lo stub è memorizzato risponde alla porta 9999 (arbitraria) su local host
- indirizzo del server è registrato al nome "server"

- ❑ **Il cliente crea un RMI connector client** configurato per connettersi al server RMI connector creato lato server:
 - ❑ URL di servizio utilizzato deve fare match con quello usato alla registrazione del servizio di connector
 - ❑ il connector client è restituito come risultato della connessione al connector server
- ❑ Il cliente ora può registrare MBean ed effettuare operazioni su di essi tramite MBeanServer remoto **in modo trasparente alla distribuzione**

Lato cliente:

```
JMXServiceURL url = new
JMXServiceURL("service:jmx:
rmi:///jndi/rmi://" +
"localhost:9999/server");

JMXConnector jmxc = JMXConnector-
Factory.connect(url, null);

MBeanServerConnection mbsc =
jmxc.getMBeanServerConnection();

mbsc.createMBean(...);
```

Piccola nota aggiuntiva, per chi eventualmente svolgerà approfondimenti su JMX...

Oltre agli usuali connettori standard RMI e RMI/IIOP, si possono utilizzare **connettori JMXMP** (ad esempio per disporre di un livello di sicurezza maggiore tramite meccanismo SSL). Per farlo, occorre effettuare il download di **JSR 160 Reference Implementation** da <http://java.sun.com/products/JavaManagement/download.html> e aggiungere il file **jmxremote_optional.jar** al classpath

- **Esempi di utilizzo di connettori JMXMP** sono inclusi in **JMX Remote API Tutorial**, reperibile insieme alla Reference Implementation di JSR 160

Esempio Semplicissimo (1)

```
/* Hello.java implementa l'interfaccia HelloMBean */
package com.example.mbeans;
public class Hello implements HelloMBean {
    public void sayHello() {
        System.out.println("hello, world");
    }
    public int add(int x, int y) {
        return x + y;
    }
    /* metodo getter per l'attributo Name. Spesso gli attributi sono
    utilizzati per fornire indicatori di monitoraggio come uptime o
    utilizzo di memoria. Sono read-only e non possono essere
    modificati tramite l'interfaccia di management */
    public String getName() {
        return this.name;
    }
    /* invece anche metodi getter e setter */
    ...
}
```



Esempio Semplicissimo (2)

```
...
/* invece anche metodi getter e setter */
public int getCacheSize() {
    return this.cacheSize;
}
/* perché synchronized? */
public synchronized void setCacheSize(int size) {
    this.cacheSize = size;
    System.out.println("Cache size now " + this.cacheSize);
}
private final String name = "My First MBean";
private int cacheSize = DEFAULT_CACHE_SIZE;
private static final int DEFAULT_CACHE_SIZE = 200;
}
```



Esempio Semplicissimo (3)

```
/* HelloMBean.java - interfaccia MBean che descrive le operazioni
e gli attributi di management: 2 operazioni (sayHello e add) e
2 attributi (Name e CacheSize) */

package com.example.mbeans;

public interface HelloMBean {
    // operazioni
    public void sayHello();
    public int add(int x, int y);
    // attributi
    public String getName();
    public int getCacheSize();
    public void setCacheSize(int size);
}
```




Esempio Semplicissimo (4)

```
/* Main.java deve semplicemente istanziare HelloWorld MBean, registrarlo e
   attendere */
package com.example.mbeans;
import java.lang.management.*;
import javax.management.*;

public class Main {
    public static void main(String[] args) throws Exception {
        // Ottiene il server MBean
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        // Costruisce ObjectName per MBean da registrare
        ObjectName name = new ObjectName("com.example.mbeans:
            type=Hello");
        // Crea istanza di HelloWorld MBean
        Hello mbean = new Hello();
        // Registra l'istanza
        mbs.registerMBean(mbean, name);

        System.out.println("Waiting forever...");
        Thread.sleep(Long.MAX_VALUE);    } }

```



Esempio con Uso di Notification (1)

```
package com.example.mbeans;
import javax.management.*;

public class Hello
    extends NotificationBroadcasterSupport implements HelloMBean {

    public void sayHello() {
        System.out.println("hello, world");
    }
    public int add(int x, int y) {
        return x + y;
    }
    public String getName() {
        return this.name;
    }
    public int getCacheSize() {
        return this.cacheSize;
    }
}

```

...



Esempio con Uso di Notification (2)

```
public synchronized void setCacheSize(int size) {
    int oldSize = this.cacheSize;
    this.cacheSize = size;

    /* In applicazioni reali il cambiamento di un attributo di solito
       produce effetti di gestione. Ad esempio, cambiamento di
       dimensione della cache può generare eliminazione o allocazione
       di entry */
    System.out.println("Cache size now " + this.cacheSize);

    /* Per costruire una notifica che descrive il cambiamento
       avvenuto: "source" è ObjectName di MBean che emette la notifica
       (MBean server sostituisce "this" con il nome dell'oggetto);
       mantenuto un numero di sequenza */
    Notification n = new AttributeChangeNotification( this,
        sequenceNumber++, System.currentTimeMillis(),
        "CacheSize changed", "CacheSize", "int", oldSize,
        this.cacheSize);

    /* Invio della notifica usando il metodo sendNotification()
       ereditato dalla superclasse */
    sendNotification(n);
} ...
```



Esempio con Uso di Notification (3)

```
...
@Override
/* metadescrizione */
public MBeanNotificationInfo[] getNotificationInfo() {
    String[] types = new String[] {
        AttributeChangeNotification.ATTRIBUTE_CHANGE
    };
    String name = AttributeChangeNotification.class.getName();
    String description = "è stato cambiato un attributo!";
    MBeanNotificationInfo info =
        new MBeanNotificationInfo(types, name, description);
    return new MBeanNotificationInfo[] {info};
}

private final String name = "My first MBean";
private int cacheSize = DEFAULT_CACHE_SIZE;
private static final int DEFAULT_CACHE_SIZE = 200;
private long sequenceNumber = 1;
}
```



Caso di Studio: JMX at work in Application Server JBoss

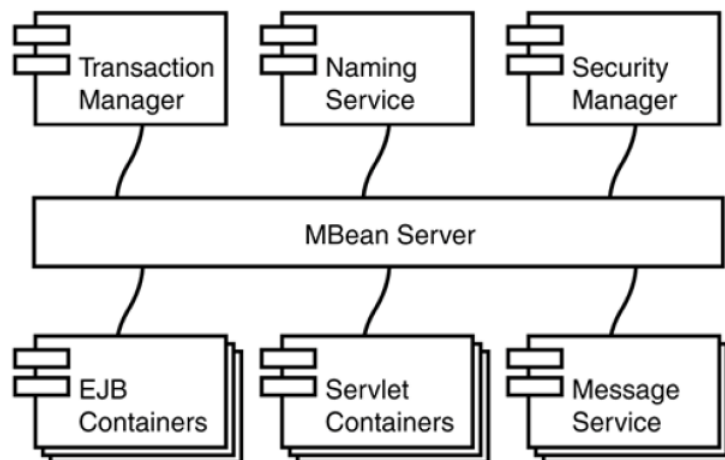
Esempio notevole: **Application server JBoss è stato costruito on top dell'infrastruttura JMX**

- ❑ **Architettura microkernel** basata su componenti MBean (**application server non-monolitico**)
- ❑ Sia le applicazioni realizzate su JBoss che l'application server sono facilmente **manageable**
- ❑ **Configurazione del server altamente flessibile**
 - Possibile scegliere fra differenti implementazioni di servizio (ad es. JMS)
 - Si può fare l'embedding di differenti container nell'application server, anche a runtime (ad es. servlet container come Tomcat, Jetty, ...)
 - Se un'implementazione di servizio non offre una funzionalità richiesta da un'applicazione (ad es. transaction manager, un determinato datasource, ...), se ne può scegliere un'altra
 - Servizi non necessari possono essere disattivati (*shut down*)



Caso di Studio: JMX at work in Application Server JBoss

- ❑ **Il nucleo dell'application server JBoss è JMX MBean server :**



- ❑ Questo rende application server estremamente semplice da **estendere con nuove funzionalità**
 - **Aggiungere nuovi servizi o componenti application-specific** si traduce nella **creazione di nuovi MBean** e nella loro registrazione al server MBean

Architettura di JBoss 5.x Application Server

Oltre all'architettura a **microcontainer** (evoluzione del bus JMX delle versioni precedenti), ritroviamo come moduli una serie di vecchie conoscenze...

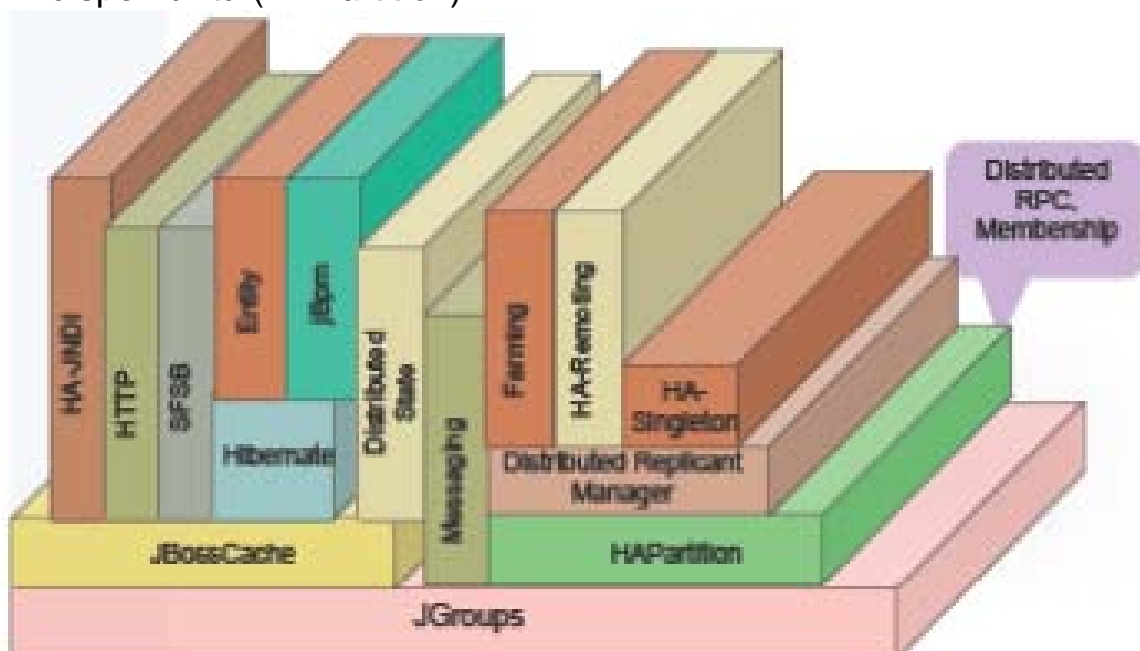
Da notare, anche per il successo di utilizzo avuto negli ultimi tempi:

- **JGroups** – framework di supporto alla realizzazione di **comunicazione multicast affidabile**



Architettura JBoss: Modularità e Livelli

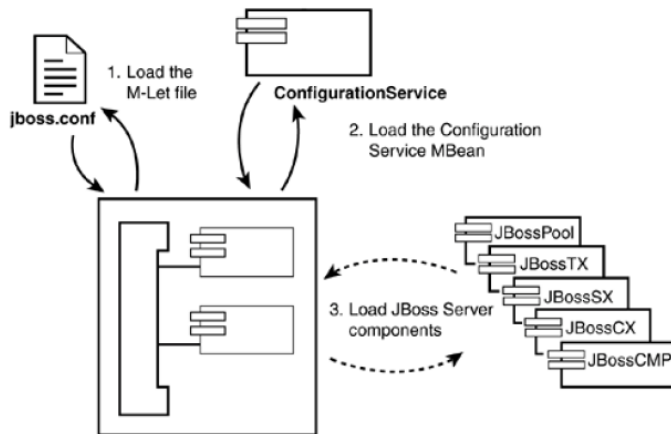
Architettura modulare e a stack, basata su comunicazione di gruppo (*JGroups*), caching (*JBossCache*) e supporto ad alta disponibilità (*HAPartition*)



Caso di Studio: JMX at work in Application Server JBoss

Esempio - uno dei componenti core caricati dal servizio M-let di JBoss è un'implementazione di ConfigurationService MBean:

- Effettua bootstrap del server
- Fa il download e configura i servizi usando il file di configurazione XML jboss.jcml



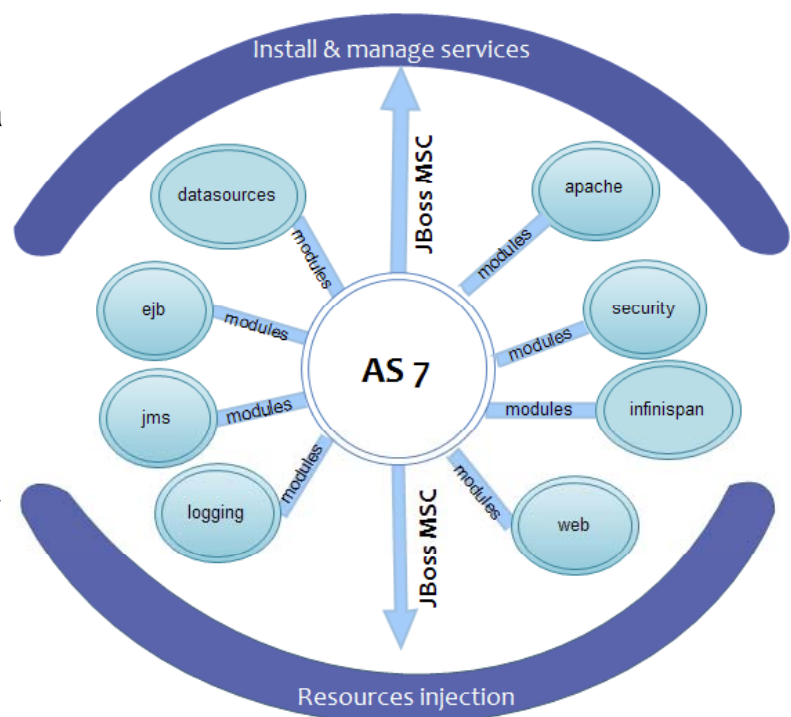
```
jboss.conf
...
<MLET
  CODE="org.jboss.configuration.
    ConfigurationService"
  ARCHIVE="jboss.jar,..../xml.jar"
  CODEBASE="..../lib/ext">
</MLET>
...

jboss.jcml
...
<mbean code="org.jboss.naming.
  NamingService"
  name="DefaultDomain:service=Naming">
  <attribute name="Port">
    1099</attribute>
</mbean>
<mbean
  code="org.jboss.naming.JNDIView"
  name="DefaultDomain:service=
    JNDIView"
/>
...
```

Oggi lo presentano così: Architettura di JBoss AS 7

Approccio modulare nel caricamento di servizi e librerie richieste in base a **metadati di dipendenza**, sia all'avvio del server che delle applicazioni (dipendenze implicite rispetto all'uso di package e file jboss-deployment-structure.xml)

Anche differenza pratica di: non un file di configurazione per sottosistema, ma unico file (standalone.xml o domain.xml)





Alcuni interessanti **riferimenti bibliografici specifici** (anche perché non citati nella bibliografia iniziale del corso; per chi eventualmente volesse approfondire nel suo progetto di approfondimento...):

- ❑ J. Lindfors, M. Fleury, The JBoss Group, **“JMX – Managing J2EE with Java Management Extensions”**, SAMS, 2002
- ❑ Sun Microsystems, Inc., **“Java Management Extensions (JMX) Technology Overview/Tutorial/Examples”**, Sep. 2004
- ❑ Sun Microsystems, Inc., **“Java Management Extensions (JMX) – Best Practices”**, 2006
- ❑ B.G. Sullins, M.B. Whipple, M. Whipple, **“JMX in Action”**, Manning, 2002
- ❑ J.S. Perry, **“Java Management Extensions”**, O'Reilly, 2002