Java in Network Management

Subodh Bapat
Sun Microsystems

(c) Copyright 1998 Subodh Bapat

Java in Network Management: Outline

- What Makes Java Especially Suitable for Network Management?
- Java in the Manager
- Java in the Agent
- Java in the Platform
- Network Management APIs in Java

Java in Network Management:

What Makes Java Especially Suitable for Network Management?

(c) Copyright 1998 Subodh Bapat

Java Features Useful to Management

- Remote Method Invocation
- Object Serialization
- Dynamic Class Loading
- Reflection
- Java Beans
- Introspection

(c) Copyright 1998 Subodh Bapat

Java Features Useful to Management

- **Remote Method Invocation**
- Object Serialization
- Dynamic Class Loading
- Reflection
- Java Beans
- Introspection

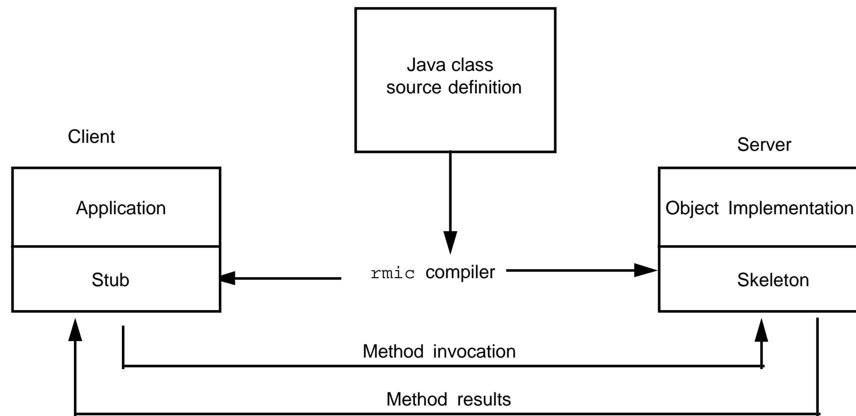
(c) Copyright 1998 Subodh Bapat

Remote Method Invocation

- Distributed object communication mechanism built into the language
- Functions as a Java-to-Java ORB
- Given a Java class, generates client-side *stubs* and server-side *skeletons*
- Application makes method call on client-side stubs, which is transparently invoked on server object implementation

(c) Copyright 1998 Subodh Bapat

RMI Architecture



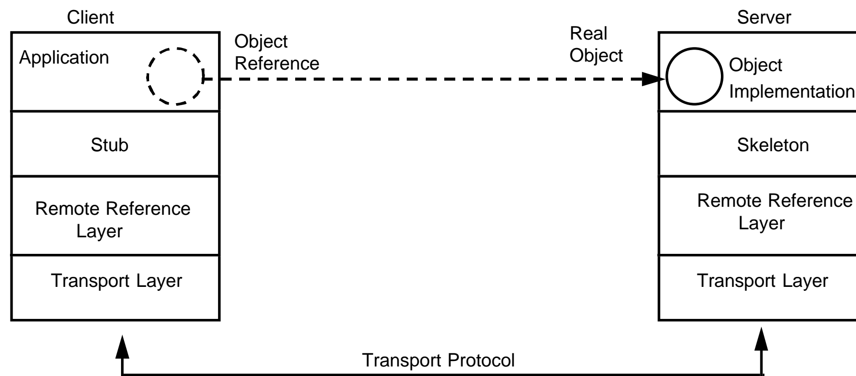
(c) Copyright 1998 Subodh Bapat

RMI Architecture

- RMI really has three layers:
  - Stub/skeleton layer
  - A Remote Reference Layer
  - Transport Layer
- An application need only interact with the interface methods on the stub
- The remote reference and the transport is automatically handled for you by the JVM.

(c) Copyright 1998 Subodh Bapat

RMI Internals



(c) Copyright 1998 Subodh Bapat

The RMI Remote Reference Layer

- This layer is responsible for:
  - Carrying our semantics of the method invocation
  - Managing the communication between the stubs/skeletons and the lower-level transport layer interface
  - Managing the reference to the remote object
  - Managing automatic connection re-establishment strategy
  - Has both client-side and server-side semantics

(c) Copyright 1998 Subodh Bapat

The RMI Remote Reference Layer

- On the client side, the Remote Reference Layer:
  – Maintains a table of known remote objects and their remote references
- On the server side, the Remote Reference Layer:
  – Hides the diffferences between objects in the server VM that are
    - are always running in the server VM
    - constructed on demand and garbage-collected when no one is using them
  – Delivers method invocation to the server objects
  – Maintains reference counts to server objects

(c) Copyright 1998 Subodh Bapat

The RMI Transport Layer

- This layer is responsible for:
  – receiving a reference from the client-side remote reference layer
  – locating the RMI server for the requested remote object
  – establishing a socket connection to the object server
  – passing the connection information back up to the client-side Remote Reference Layer
  – adding this remote object to the list of known remote objects that it is communicating with (so that connection establishment is avoided for a second reference to the same object)
  – monitoring connection liveness

(c) Copyright 1998 Subodh Bapat

The RMI Transport Layer

- Object name resolution and transport is executed using four basic abstractions:
  - *Connection*: The name given to the entire abstraction. Each abstract connection consists of a *channel*, at least two *endpoints*, and a *transport*.
  - *Endpoint*: Used to denote either an address (if in the local VM) or the address of a remote JVM. An endpoint can be uniquely mapped to a transport.
  - *Channel*: Used as a conduit between two address spaces. Responsible for managing connections between the local address space and the remote address space.
  - *Transport*: Used as the conveyance mechanism for a specific channel. For a method invocation, receives downcalls from the remote reference layer on the client side, and makes upcalls into the remote reference layer on the server side.

(c) Copyright 1998 Subodh Bapat

RMI Distributed Garbage Collection

- Handles by the transport layer
- Based on a reference-counting strategy for server objects
- A server object has two kinds of references:
  - a *live* reference when there is a remote client reference to an object
  - a *weak* reference when there are no remote client references to an object; the object may be discarded if there are no local references to it either.

(c) Copyright 1998 Subodh Bapat

RMI Distributed Garbage Collection

- At startup, the server implementation constructs an object that has a weak reference
- When the client requests an object, the client’s JVM creates a live reference to the stub object
- When the first method is invoked on the object, the client JVM sends a “referenced” message to the server JVM
- When the object goes out of scope on the client, an “unreferenced” method is sent to the server JVM
- When the remote reference count on the server object drops to zero and there are no local references to it, the object may be garbage collected by the server’s VM in due course.

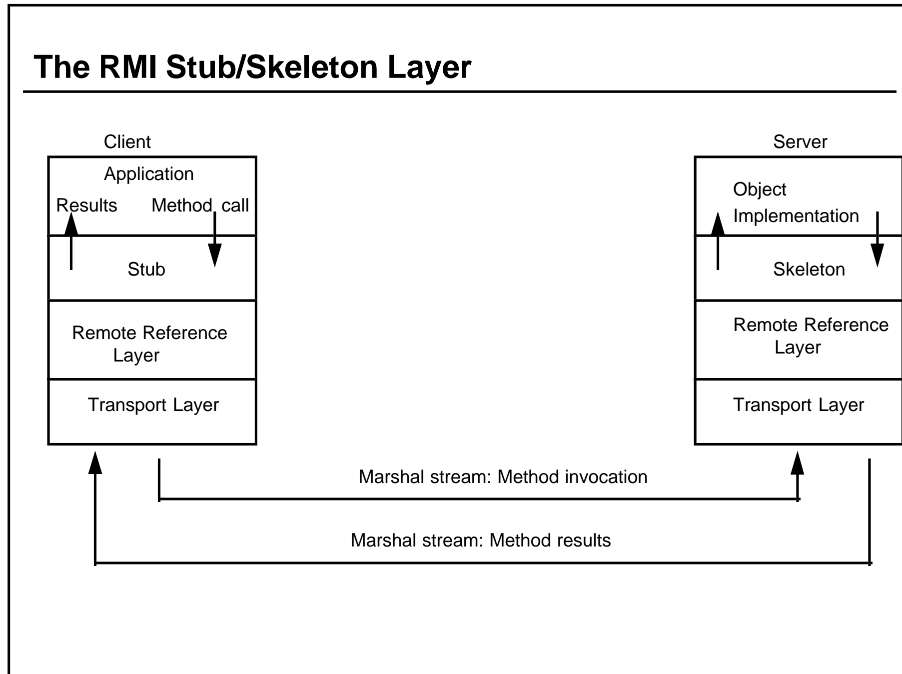
(c) Copyright 1998 Subodh Bapat

The RMI Stub/Skeleton Layer

- Is the interface between the application and the transparent distributed object system built into Java
- Does not deal with any transport specifics; simply transmits data to the Remote Reference Layer
- The stub acts as a *proxy* on the client machine for the real object implementation on the server machine
- Client applications initiating a method invocation do so on the stub object
- Server implementations servicing a method invocation do so on the skeleton object

(c) Copyright 1998 Subodh Bapat

The RMI Stub/Skeleton Layer



(c) Copyright 1998 Subodh Bapat

The RMI Stub Layer

- The client stub fields a method invocation and initiates a server-side call
- The client Remote Reference Layer returns a special I/O stream, called a marshal stream
- The marshal stream is used by the stub to communicate with the server's Remote Reference Layer
- The stub makes a remote method invocation, passing any objects to the stream
- When the client receives the results, the client RRL passes the method's return value to the stub
- The stub sends an acknowledgement to the RRL that the remote method invocation is complete.

(c) Copyright 1998 Subodh Bapat

The RMI Skeleton Layer

- The RMI skeleton on the server receives the remote method invocation on the marshal stream
- The skeleton *unmarshals* (receives and decodes) the parameters to the method call from the marshal stream
- The skeleton upcalls into the object implementation
- The skeleton receives the response from the method invocation, and *marshals* (encodes and transmits) the return value on the I/O stream

(c) Copyright 1998 Subodh Bapat

Creating an RMI Network Management Application

- RMI can be used to create Java applications that run on a “thin client” system and talk to a network management platform
- Application classes specific to network management can be defined and compiled through `rmic`
- Lightweight applications can then make RMI calls to object implementations on heavyweight platforms

(c) Copyright 1998 Subodh Bapat

Practical Tips for an RMI Management Application

- Define application-specific classes (e.g. `Alarm`, `Device`, etc.)

```
interface Device {
    public DeviceName getName();
    public Alarm[] getAlarms();
}

interface Alarm {
    public AlarmType getAlarmType();
    public TimeStamp getTime();
    public Severity getSeverity();
    public void acknowledge();
    public boolean IsAcknowledged();
    public void clear();
    public boolean IsCleared();
}
```

(c) Copyright 1998 Subodh Bapat

Creating an RMI Network Management Application

- Code the remote objects to be operated on as Java `interfaces`
- Code the implementation classes for these interfaces (`AlarmImpl`, `DeviceImpl`)
- Compile the interface and implementation classes
- Generate stub and skeleton classes using `rmic` on the implementation classes
- Code a server application to link up the skeletons and the `Impl` classes
- Code a client application to make invocations on the stub objects
- Compile both applications
- Start the `rmiregistry` and the server application
- Bring up the client and invoke methods on it

(c) Copyright 1998 Subodh Bapat

RMI Network Management Applications: Trade-Offs

- Trade-off is between how thin you want to make the client and how many network calls you want to make
- For example, is it worth it to make a network call for each of the methods `getTime()`, `getSeverity()` etc. on `Alarm` objects, or should we cache some of this information locally on the client? (If so, we can optionally remove these methods from the remote interface).
- Trade-offs must be carefully decided based on performance requirements and available memory in the thin client
- Any class that has methods that are real remote method invocations must extend the `Remote` interface (in package `java.rmi.Remote`), and must be declared `public` in order to be accessible remotely
- Any method that makes a real network call must be declared to throw a `RemoteException` (in case there are any problems accessing the server)

RMI Network Management Applications: Trade-Offs

- Examples of classes where some methods are network calls and some methods are simply local invocations:

```
import java.rmi.*;

public interface Device extends Remote {
   public DeviceName getName() throws RemoteException;
   public Alarm[] getAlarms() throws RemoteException;
}
public interface Alarm extends Remote {
   public boolean loadAlarmData() throws RemoteException;
   public AlarmType getAlarmType();
   public TimeStamp getTime();
   public Severity getSeverity();
   public void acknowledge() throws RemoteException;
   public bool IsAcknowledged() throws RemoteException;
   public void clear() throws RemoteException;
   public bool IsCleared() throws RemoteException;
}
```

RMI Network Management Applications: Server Side

- Server side `Impl` classes must be declared to implement the client-side interface
- Must provide appropriate constructors
- Must either extend `UnicastRemoteObject` (this indicates the implementation is a singleton server that uses RMI's default socket-based transport), or must explicitly export itself as remote by calling `java.rmi.server. UnicastRemoteObject.exportObject()`
- May provide a `finalize()` method to perform cleanup for garbage collection

(c) Copyright 1998 Subodh Bapat

RMI Network Management Applications: Server Side

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class DeviceImpl
    extends UnicastRemoteObject
    implements Device {
    private DeviceName name;
    private boolean checkNameInUse();
// construct the Device object
    public DeviceImpl (DeviceName deviceName)
                    throws RemoteException
    { if (checkNameInUse()) throw RemoteException
          else name = deviceName; }
    public DeviceName getName() { return name };
    public Alarm[] getAlarms() {
          // make JDBC call into device database to
          // get device-specific alarms
    }
}
```

(c) Copyright 1998 Subodh Bapat

The RMI Registry

- The server “publishes” the instance of the `DeviceImpl` object by binding the object, after it is instantiated, to a name that is stored in the `rmiregistry`.
- The binding occurs in the main server application using the `Naming.rebind()` call:

```
DeviceName myRouterName =
           new DeviceName(“xyz”,“Cisco 7000”);
DeviceImpl myRouter = new DeviceImpl (myRouterName);
Naming.rebind
           (“rmi://myDeviceServer:1099/RouterXYZ”,
                                    myRouter);
```

(c) Copyright 1998 Subodh Bapat

The RMI Registry

- Each registered object is named by a URL of the form

```
rmi://hostName:portNumber/objectName
```

- The methods `Naming.bind()` and `Naming.rebind()` add an entry to the `rmiregistry` (the difference being that `Naming.bind()` throws `java.rmi.AlreadyBoundException`)
- Clients find the object by using the `Naming.lookup()` call
- RMI registry must be running before the server application binds a name
- For security reasons, registry must run on same host as server
  - prevents clients from changing a server’s remote registry
  - permits clients to look it from any host

(c) Copyright 1998 Subodh Bapat

What Makes RMI Especially Suitable for Network Management?

- Built-in distributed object model
- Built-in generation of thin-client fat-server counterparts
- Invocations of network management operations on a Java object on a client can be transparently referred to their implementations on a management platform
- Choice of how much processing you want to do locally in the client and which operations you want to refer to the platform

(c) Copyright 1998 Subodh Bapat

Java Features Useful to Management

- Remote Method Invocation
- **Object Serialization**
- Dynamic Class Loading
- Reflection
- Java Beans
- Introspection

(c) Copyright 1998 Subodh Bapat

Object Serialization

- RMI uses object serialization to pass objects that are arguments to method invocations and objects that are returned results
- Serialization encodes an object's data values according to certain rules and puts them in a marshal stream
- Object structures are maintained when saved
- Serialization does a *deep copy*:
  - traversing referenced objects
  - copies data values from referenced objects
  - smart enough to compute finite transitive closures even if objects contain circular references
- Fields that are declared `static` or `transient` are not written
- Does not include class metadata in serialized objects

(c) Copyright 1998 Subodh Bapat

Object Serialization

- Can be used for making individual objects persistent (similar to OODBMS systems)
- Can be used to save the entire state of running programs in a file (like a core file, except can restart the program from the saved file!)
- Excellent medium-weight solution for sending objects over a network
- Used by RMI and Java Beans APIs for storing objects, sending objects over a network, and communicating with objects

(c) Copyright 1998 Subodh Bapat

Object Serialization

- Any object that is to be serialized must implement either:
  - the `Serializable` interface (which use the built-in encoding rules)
  - the `Externalizable` interface (which permits the use of your own encoding rules)
- Most Java Beans use `Serializable`
- An `Externalizable` object can be written out in any needed data format; but the programmer has to do all the work
  - methods `readExternal()` and `writeExternal()` must be given your own implementations
  - new methods can be added

(c) Copyright 1998 Subodh Bapat

Object Serialization and ASN.1 Encoding

- Java's built-in serialization mechanism is, on the average, *faster* than an ASN.1 encoder
- Java's built-in serialization rules produce encodings which are, on the average, *more compact* than ASN.1 BER
- Advantages of Java serialization:
  - easier and more natural to program
  - permits programmers to think in terms of distributed objects, rather than worrying about exact format and content of PDUs
- Disadvantages of Java serialization:
  - can be slower if the object being serialized is complex and has many inter-object references
  - less control over exact format and content of PDUs, compared to ASN.1 definitions

(c) Copyright 1998 Subodh Bapat

Using Externalization for ASN.1 Encoding

- An `Externalizable` class has object read/write methods implemented in the class itself rather than in the `FileInputStream` and `FileOutputStream` objects:

```
public interface ASN1Externalizable
          extends Externalizable {
    public void writeExternal(ObjectOutput out)
          throws IOException, ASN1EncoderException;
    public void readExternal(ObjectInput in)
          throws IOException, ASN1DecoderException;
    public void writeExternalAsPER(ObjectOutput out)
          throws IOException, ASN1EncoderException;
    public void readExternalAsPER(ObjectInput in)
          throws IOException, ASN1DecoderException;
}
```

(c) Copyright 1998 Subodh Bapat

Object Serialization and ASN.1 Encoding

- An object can use built-in Java serialization or implement its own ASN.1 encoding/decoding capabilities for its own fields:

```
class AlarmData implements ASN1Externalizable {
     public AlarmType alarmType;
     public TimeStamp timestamp;
     public Severity  severity;
     public boolean   isAcknowledged;
     public boolean   isCleared;
     public void writeExternal(ObjectOutput out) {
             // put calls to ASN.1 encoder here
     }
     public void readExternal(ObjectInput in) {
             // put calls to ASN.1 decoder here
     }
}
```

(c) Copyright 1998 Subodh Bapat

What Makes Object Serialization Especially Suitable for Network Management?

- Provides a built-in mechanism to transfer management operation data (operation arguments and return types) over distributed objects in a network
- If managers and agents both use Java RMI to communicate instead of a standard network management protocol, object serialization becomes an alternative to ASN.1 encoding and decoding
- Can be customized to do ASN.1 encoding and decoding (using either BER, PER, DER, etc.) if necessary

(c) Copyright 1998 Subodh Bapat

Java Features Useful to Management

- Remote Method Invocation
- Object Serialization
- **Dynamic Class Loading**
- Reflection
- Java Beans
- Introspection

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading: Local Loading

- The default class loader is used to load classes from the local `CLASSPATH` (for example, the class that calls `main()` must be loaded first before anything can happen).
- All classes referenced by the class that calls `main()` are loaded by the default class loader from the local `CLASSPATH`
- Remote class loading using `RMIClassLoader` is used for those classes needed for remote method invocation

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading: Remote Loading

- Java permits applications to dynamically remotely load new class definitions
- Clients may download classes from a server-defined codebase
- Servers may download classes from client-supplied URLs
- On the server-side JVM, two properties decide where clients may download classes from:
  - `java.rmi.server.codebase`: This property is a URL that indicates from where clients may download classes
  - `java.rmi.server.useCodebaseOnly`: This property is a boolean which, if set to `true`, will disable the loading of classes by the server from client-supplied URLs.

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading

- If the client program is an applet (i.e. running in a browser), then it is dependent on the browser's VM, and must use the `SecurityManager` and `RMIClassLoader` being used by the browser
- If the client program is an application, classes related to all its RMI activities will be automatically downloaded for it from the RMI server:
  - remote interface definitions
  - stub classes
  - parameter classes (classes that are arguments to and return values from remote methods)
  - URL for the loadable classes is encoded in the marshal stream (this is usually either the URL of any non-default class loader that has been installed, or that of the defined `codebase`).

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading

- To load additional classes from the server, the client must use `RMIClassLoader.loadClass()`, providing as an argument an RMI Naming URL.
- Classes are loaded using Java's architecture-neutral distribution format (bytecodes)
- Classes are transmitted on the wire as ordinary data that is serialized
- No special configuration is required for the client to send RMI calls through firewalls
- If a direct socket connection cannot be made to the server, the RRL will retry the request via HTTP by sending the RMI call as an HTTP `POST` request (this often gets it through firewalls)
- If classes cannot be loaded, the appropriate exception will be thrown

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading: Security Aspects

- A `SecurityManager` is a Java class that enforces restrictions on a program's capabilities, including
  - local file access
  - thread access
  - access to system information
  - dynamic class loading
- There can be at most one `SecurityManager` per VM, and can be installed only once. Hence
  - the `SecurityManager` cannot be disabled and is not garbage collected
  - the `SecurityManager`, once installed, cannot be reinstalled (either by user code or by dynamic class loading)

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading: Security Aspects

- An application may install its own security manager by inheriting from `java.lang.SecurityManager`

```
System.setSecurityManager (new mySecurityManager);
```

- The behavior of the default `SecurityManager` is that everything fails
- If an application installs no security manager, almost everything is allowed, except loading of classes (either local or remote)
- To permit network management applications (in either a manager or an agent) to dynamically load classes, the `SecurityManager` must be configured to permit dynamic class loading

(c) Copyright 1998 Subodh Bapat

What Makes Dynamic Class Loading Especially Suitable for Network Management?

- Linkers are eliminated
- Only necessary code is loaded
- The latest version of the managed object behavior is loaded on demand at run-time, not link time
- Network management applications can be distributed but still use common services
- Not necessary to relink entire project when one module changes
- Flexibility to change metadata in managers and agents

(c) Copyright 1998 Subodh Bapat

Java Features Useful to Management

- Remote Method Invocation
- Object Serialization
- Dynamic Class Loading
- **Reflection**
- Java Beans
- Introspection

(c) Copyright 1998 Subodh Bapat

Reflection

- The Java language provides built-in metadata for each Java object
- Generalization of Run-Time Type Identification with a complete metadata API
- Java Core Reflection API defines the reflection methods that can be invoked on any Java object
- Reflection can be used to determine the structure and behavior of an unknown Java object
- Reflection can be used to exercise the behavior of an unknown Java object (invoke a method whose signature has been discovered through reflection)
- Both the Java Beans and Object Serialization APIs use the Core Reflection API to obtain information about an object

(c) Copyright 1998 Subodh Bapat

Core Reflection API

- The following information can be obtained from any Java object:
  - `Class`
  - `Interface`
  - `Method`
  - `Field`
  - `Constructor`
  - `Array`
  - `Modifier`
- All of the above implement the `Member` interface

(c) Copyright 1998 Subodh Bapat

Reflection: The Member Interface

- The `Member` interface encapsulates the common metadata features of any member of any class

```
public interface Member {
    // obtain enclosing class or interface
    public Class getDeclaringClass();
    // obtain member name
    public String getMemberName();
    // obtain Modifiers
    public int getModifiers();
}
```


Reflection: Using Class Information at Runtime

- Runtime extraction of class definition and constructor
- The method `this.getClass()` (or the data member *`classname`*`.class`, if you don't have an instantiated object of that class) or the returns the metadata `Class` object

```
Device myDevice = new Device("/systemId=\"HQ_NOC\"/\
                          deviceId=\"CiscoRouter5\"");
Class deviceClass = myDevice.getClass();
```

- Obtain the constructor that constructs a `Device` from a `String` argument

```
Class[] stringSignature = {String.class};
Constructor ctor =
          deviceClass.getConstructor (stringSignature);
```

Reflection: Using Class Information at Runtime

- Invoke the constructor that you just determined with an actual argument to get a new instance of the same class:

```
String[] actualArgsForNewInstance =
    {"/systemId=\"HQ_NOC\"/deviceId=\"CiscoRouter6\""};

Device newDevice = (Device) ctor.newInstance
                        (actualArgsForNewInstance);
```

- Since an object can be used to make other objects just like itself, an object can be used as the factory for other objects

(c) Copyright 1998 Subodh Bapat

Reflection: Methods

- An object's class can be asked for all the methods and constructors it has
- Example: Use reflection to see if the `mySwitch` object has a `reboot()` method

```
Method[] allMethods = mySwitch.getClass().getMethods();
for (i=0; i<allMethods.length; i++) {
    if (allMethods[i].toString() == String("reboot"))
           { Method rebootMethod = allMethods[i]; }
}
```

- An application may invoke methods on the object that it has discovered through reflection

```
rebootMethod.invoke (mySwitch, null);
```

(c) Copyright 1998 Subodh Bapat

Reflection: Other Features

- A `Class` object can be checked to see if it is an interface or a real class
- An `Interface` object can be checked to see what other interfaces it extends
- A `Class` can be checked to see if it implements a particular interface
- All features and capabilities available from the core reflection API can be determined by simply reflecting on the `java.lang.Class` class

(c) Copyright 1998 Subodh Bapat

What Makes Java Reflection Especially Suitable for Network Management?

- Dynamic access to metadata built into the language
- For managed objects which are represented by specific mapped Java classes, reflection can be used to determine metadata at run-time
- For managed objects which are represented by generic Java classes, reflection can be customized to yield management-information-model-specific metadata

(c) Copyright 1998 Subodh Bapat

Java Features Useful to Management

- Remote Method Invocation
- Object Serialization
- Dynamic Class Loading
- Reflection
- **Java Beans**
- Introspection

(c) Copyright 1998 Subodh Bapat

Java Beans

- JavaBeans is a portable, platform-independent component model for Java applications
- Beans permit programmers to create software pieces called *components* which can be combined together (with little or no programming) to create applications
- A Java Bean can be defined as “a reusable software component that can be combined with other components in a visual application builder tool to create software applications”.

(c) Copyright 1998 Subodh Bapat

Features of Java Beans

- *Introspection*: Using the core Java Reflection API, a Bean's metadata can be interrogated, permitting other Beans to discover its capabilities
- *Customization*: Enables a developer to customize a Bean's appearance and behavior during the application development phase
- *Events*: Beans may notify other Beans about events that occur via a standard Bean event distribution mechanism
- *Properties*: When established, enables application developers to program the choices for the appearance and behavior of a Bean
- *Persistence*: Enables the storage and restoral of a Bean's state in a standard way

(c) Copyright 1998 Subodh Bapat

Java Beans vs. Java Classes

- A Bean consists of one or more Java classes
- All Java classes are not Beans
- A Java Bean is a Java class that *must follow* the naming conventions specified in the JavaBeans API specification for properties, methods, events, etc.
- Visual application development environments (e.g. Java Studio) depend on these naming conventions to be able to discover a Bean's properties, methods, events etc.
- If a Bean does not conform to naming conventions, these application builders will not be able to learn about it, and hence will be unable to hook it up to other Beans to create more complex behavior and functionality

(c) Copyright 1998 Subodh Bapat

Bean Interfaces

- Beans can be combined with other Beans to create applications via the *interface publish and discovery* mechanism in visual application development environments
- Example: The value of a property in one Bean (a *target property*) can be set to automatically reflect the value of another property in another Bean (the *source property*)
- Example: A Bean that generates events can be configured to automatically send the event to another Bean
- All of the above can be accomplished using drag-and-drop in a visual builder tool, without any programming

(c) Copyright 1998 Subodh Bapat

The Java Bean Event Model

- Each Bean event has a *source* which originates or *fires* the event
- Each event may have multiple *listeners* which want to be notified of events of a particular type
- The event listener indicate their interest in the event by *registering* with the event source

(c) Copyright 1998 Subodh Bapat

Bean Event Registration and Firing

- An event listener registers with an event source with with an `add`*`EventName`*`Listener(`*`EventName`*`Listener l)` method
- An event listener deregisters with an event source with a `remove`*`EventName`*`Listener(`*`EventName`*`Listener l)` method
- For each event type, the interface *`EventName`*`Listener` must be defined, and implemented by all registered listener objects
- If the event source is a unicasting source, it permits only one listener to be registered at any time
- The source Bean notifies its listeners of an event by constructing an event object, iterating through its registered listeners, invoking the methods on their *`EventName`*`Listener` interfaces, passing in the event object as an argument.

(c) Copyright 1998 Subodh Bapat

Bean Properties

- A Bean property is a named attribute of a Bean that can affect its behavior.
- Properties may be:
  - *Simple*: changes in this property don't affect anything else
  - *Bound*: changes in this property result in a notification being sent to another Bean
  - *Constrained*: changes in this property need to be validated by another Bean, and may be vetoed

(c) Copyright 1998 Subodh Bapat

Bean Property Naming Conventions

- A visual application builder tool recognizes a Bean property if its methods follow certain naming conventions
- If the Bean has the method `getAbc()`, it is considered to have the property `Abc` as a readable property

```
public PropertyType getPropertyName();
```

- If the Bean has the method `setAbc()`, it is considered to have the property `Abc` as a writable property

```
public void setPropertyName (PropertyType a);
```

- The Bean introspector depends on these naming conventions in order to recognize Bean properties

(c) Copyright 1998 Subodh Bapat

Bound Properties

- Bound properties must have:
  - registration methods for `PropertyChangeListeners`
  - the ability to fire `PropertyChangeEvents`

```
public void addPropertyChangeListener
                  (PropertyChangeListener l) {...}

public void removePropertyChangeListener
                  (PropertyChangeListener l) {...}
```

- Within the set*PropertyName* method, the method `firePropertyChange()` must be called to let all registered listeners know of a change in the object's value:

```
support.firePropertyChange (String PropertyName,
                  Object oldValue, Object newValue);
```

(c) Copyright 1998 Subodh Bapat

Constrained Properties

- Changes to a constrained property result in notifications being fired to registered `VetoableChangeListener` objects
- If a listener objects to this change, it throws a `PropertyVetoException`
- The source Bean is responsible for catching this exception and reverting back to the old value of the property
- The naming convention for a constrained property is the same as that of other properties, except the `setXXX` method must be declared to throw a `PropertyVetoException`:

```
public void setPropertyName (PropertyType a)
                        throws PropertyVetoException;
```

- If the source Bean reverts the property back to its old value, it must still fire a new notification to all its listeners

(c) Copyright 1998 Subodh Bapat

Java Features Useful to Management

- Remote Method Invocation
- Object Serialization
- Dynamic Class Loading
- Reflection
- Java Beans
- **Introspection**

(c) Copyright 1998 Subodh Bapat

Bean Introspection

- Introspection is similar to Java reflection, except it applies to Bean properties and events in a Java Bean
- Uses a class called `Introspector` (in package `java.beans`)
- Fills out Descriptor classes
- Primary method of `Introspector` class is `getBeanInfo()`, which returns `BeanInfo` objects
- Beans must follow a naming convention in which the information about itself is provided in a separate *beanName*`BeanInfo` class
- Naming convention for properties are specified as a set of design patterns
- The `Introspector` uses the Core Reflection API

(c) Copyright 1998 Subodh Bapat

Bean Naming Conventions: Properties

- Simple Properties

```
public PropType getPropName();
public void setPropName (PropType a);
```

- Boolean Properties

```
public boolean isPropName();
public void setPropName (boolean a);
```

- Indexed Properties

```
public PropElement getPropName(int index);
public void setPropName (int index, PropElement e);
public PropElement[] getPropName();
public void setPropName (PropElement[] eArray);
```

(c) Copyright 1998 Subodh Bapat

Bean Naming Conventions: Events

- Event listener registration for multicast events

```
public void addEventNameListener(EventNameListener l);
```

- Event listener registration for unicast events

```
public void addEventNameListener(EventNameListener l)
    throws java.util.TooManyListenersException;
```

- Event listener deregistration

```
public void removeEventNameListener (
                              EventNameListener l);
```


Bean Introspection

- Introspector can be called after the Bean is instantiated

```
beanName = "myDevicesPackage.\
                        remoteAccessServers.Xylogics";
myXylogicsBean = (Component) Beans.instantiate
                        ( classLoader, beanName );
Class beanClass = myXylogicsBean.getClass();
BeanInfo xylogicsInfo =
           Introspector.getBeanInfo(beanClass);
```

- Or, on the class itself, before anything is instantiated:

```
beanName = "myDevicesPackage.\
                        remoteAccessServers.Xylogics";
Class beanClass = Class.forName(beanName);
BeanInfo xylogicsInfo =
           Introspector.getBeanInfo(beanClass);
```

Bean Analysis Using BeanInfo

- A `BeanInfo` object can be analyzed in a visual application development environment
- A Bean developer can annotate the `BeanInfo` to do many useful things, such as:
  - limit it to display only a few relevant properties, instead of everything that has a get and set method
  - provide a visual representation for the Bean (GIF images, icons, etc.)
  - add descriptive names to properties
  - specify additional "smart" customizer classes

(c) Copyright 1998 Subodh Bapat

The BeanInfo Interface

- The `SimpleBeanInfo` interface permits application development environments to inspect the Bean, using the following methods:
  - `getBeanDescriptor()`
  - `getAdditionalBeanInfo()`
  - `getPropertyDescriptors()`
  - `getDefaultPropertyIndex()`
  - `getEventSetDescriptors()`
  - `getDefaultEventIndex()`
  - `getMethodDescriptors()`
  - `getIcon()`

(c) Copyright 1998 Subodh Bapat

Customizing BeanInfo

- All methods in the `SimpleBeanInfo` interface can be overridden to provide customization
- For example, Bean properties can be annotated by overriding the default `getPropertyDescriptor()` method in the corresponding `BeanInfo` class
- This affects how properties appear when they are visually displayed

(c) Copyright 1998 Subodh Bapat

Annotating Properties with BeanInfo

```
import java.beans.*;

 public class XylogicsBeanInfo extends SimpleBeanInfo {
   public PropertyDescriptor[] getPropertyDescriptors() {
     PropertyDescriptor pd = null;
     try {
       pd = new PropertyDescriptor( "possibleSpeeds",
                                   XylogicsBean.class);
     } catch (IntrospectionException e) {
       System.err.println("Introspection exception\
                                          caught: " + e);
     }
     pd.setDisplayName("Possible Speeds:\
                         9.6K, 14.4K, 28.8K, 33.6K, 56K");
     PropertyDescriptor result[] = { pd };
     return result;
   }
 }
```

(c) Copyright 1998 Subodh Bapat

Other Uses of BeanInfo

- The method `getAdditionalBeanInfo()` can be customized to provide only incremental changes to `BeanInfo`, leaving the rest of the properties to be returned as per their default values
- In the previous example, we overrode `getPropertyDescriptors()`, thereby destroying all other properties of the `XylogicsBean` and leaving only the `possibleSpeeds` property visible
- If we had overridden `getAdditionalBeanInfo()` instead, all the other default properties would have still been visible, and `possibleSpeeds` would have been returned as an additional property
- Similarly, `getMethodDescriptors()` can be overridden to hide any trivial housekeeping methods you don't want displayed in a visual app builder tool, showing only the really meaningful methods

(c) Copyright 1998 Subodh Bapat

Other Java Bean Customization

- Customized property editors and property sheets can be added for any Bean component
- Customized event adapters can be added to any Bean:
  - multiplexing adapters
  - demultiplexing adapters
- Beans can be used with RMI to obtain a distributed component environment
- Coming soon: InfoBus technology from Lotus/IBM, which permits Java Beans to be hooked up with each other using dynamically defined information (e.g. bound properties need no longer be statically defined in a class).

(c) Copyright 1998 Subodh Bapat

What Makes Java Beans Especially Suitable for Network Management?

- Component object model can be used to develop network management components
- Managed objects can be easily mapped as Java Beans
- Management platform services can be easily mapped as Java Beans
- Management applications can be rapidly developed by connecting managed object beans and service beans together in a visual application development environment
- Leads to development of management applications with minimal or no programming

(c) Copyright 1998 Subodh Bapat

Java in Network Management:

Java in the Agent

(c) Copyright 1998 Subodh Bapat

Benefits of Java-Based Agents

- Enable rapid agent application development, using productivity benefits of Java
- Agent features and functions can be represented as Java Beans, allowing visual Bean development environments to create agent applications with minimal programming
- Dynamic downloads of agent behavior using Java distribution technology
- Agent software upgrade problem is minimized, hence configuration management of large networks becomes easier.
- Software can be upgraded using either “manager push” technology or “agent pull” technology

(c) Copyright 1998 Subodh Bapat

Java-Based Agents

- Smart agents can be developed with sophisticated functionality and behavior
- Depending on the implementation of Java classes, much of the processing can be done within agent services
- If the agent has sophisticated local processing capabilities, the manager can be offloaded and the network traffic reduced
- The ability to dynamically download capabilities into an agent means the local intelligence in an agent can be changed over time
  - dumb agents can be made smarter
  - smart agents can be made dumber

(c) Copyright 1998 Subodh Bapat

Architecture of a Java-Based Agent

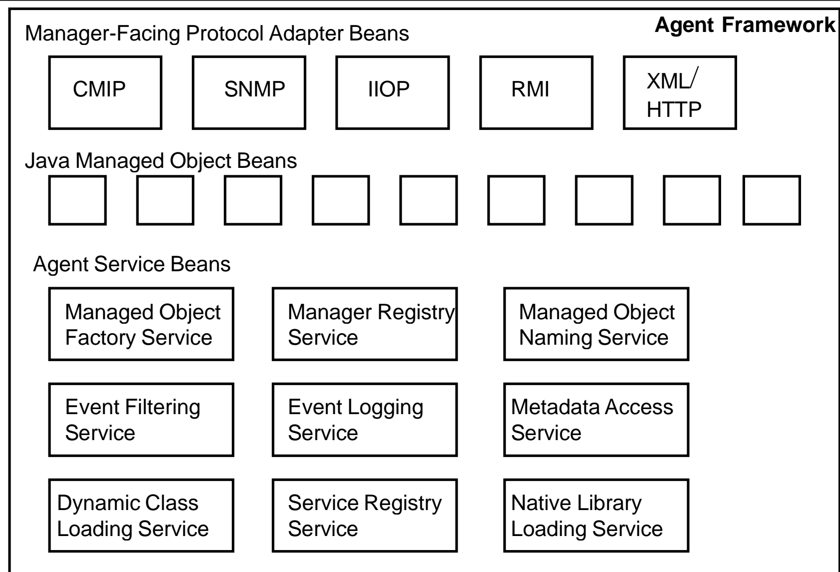
- Java Virtual Machine
- The following Java Bean objects in a virtual machine:
  - A Java-based *Agent Framework*
  - Java *managed objects*
  - An *Agent Naming Service* to look up instances of Java managed objects
  - Java-based SMI *metadata*
    - generic managed objects for non-Java SMI need a metadata database
    - metadata for managed objects defined in Java SMI, or specific-mapped managed objects from a non-Java SMI, is available via class reflection or Bean introspection

(c) Copyright 1998 Subodh Bapat

Architecture of a Java-Based Agent

- *Manager-Facing Protocol Adapter* services (optional, for multilingual agents): SNMP, CMIP, RPC, RMI, HTTP/XML, IIOP, SSL, etc.
- *Agent services* (optional):
  - event filtering/discrimination
  - logging
  - metadata
  - access control
  - event generation
  - persistence
  - relationship management
  - dynamic class loading
  - dynamic native library loading

Architecture of a Java-Based Agent Framework



The Java Agent Framework

- Is a singleton class per Java Agent
- Provides a place to hook in agent services
- Provides a starting point to to interrogate the *Agent Naming Service* to look up Java Managed Objects
- Provides a starting point for accessing SMI *metadata*
- Provides a *service registry service* to dynamically add new services in the agent
- Provides an inter-service communication framework for agent services

(c) Copyright 1998 Subodh Bapat

Java Managed Object Beans

- A Java *managed object bean* is a software abstraction of a resource that is controlled and monitored by an agent
- If implemented as a Java Bean, the Java managed object can be visually manipulated in a Bean development environment
- The Java managed object has methods to
  - manipulate managed object attribute values
  - emit notifications
  - perform actions (arbitrary methods to execute behavior)
  - assign a name for the managed object and register it with the agent's Managed Object Naming Service

(c) Copyright 1998 Subodh Bapat

Java Managed Object Beans

- Any of the following cases may occur:
  - The Java managed object bean may execute operations invoked on it (get/set attributes, run actions) by communicating with some real resource in an implementation-specific manner.
  - The Java managed object bean may locally implement some attributes for which the real resource has no notion or representation (e.g. an administrativeState attribute or equivalent).
  - The Java managed object bean may locally cache values for attributes already available in the real resource, for faster access.
  - The Java managed object bean may locally implement algorithms to compute attribute values from other information available in the real resource.
  - The Java managed object bean may itself be the real resource and implement all its attributes, actions, and behaviour as Bean properties and methods.

(c) Copyright 1998 Subodh Bapat

Protocol Operations on Java Managed Object Beans

- Network management protocol operations can easily be mapped onto Bean operations:
  - Object creation: `Beans.instantiate()`
  - Object deletion: Nothing (drop reference, Bean goes out of scope, is garbage collected)
  - Get attribute: `getXXX()` method on Bean properties (specific) or `getAttribute("XXX")` method (generic)
  - Set attribute: `setXXX(value)` method on Bean properties (specific) or `setAttribute("XXX", value)` method (generic)
  - Actions: call appropriate methods on Bean
  - Notification emission: Fire events to listeners using Bean Event Model (Protocol adapter Beans can be Bean event listeners, and translate Bean events to management protocol events that leave the VM)

(c) Copyright 1998 Subodh Bapat

Java Behavior in Java Managed Objects

- Java behavior can be implemented for Java Managed Objects in a Java agent
- This can be done using the Dynamic Class Loading Service
- If the invocation of a method (e.g. to get or set an attribute or to perform an action) references a class that is not currently loaded, the Dynamic Class Loading service can load the appropriate class
- This essentially happens for free (since it is a characteristic of the programming language)

(c) Copyright 1998 Subodh Bapat

Service Beans in a Java Agent

- An *Agent Service* is a Java Bean that implements a particular service in an agent
- An agent service is similar to a Platform Service, except that the scope of the service is limited to a single Java agent
- It may support the same interface(s) as a platform service

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading Service

- Loads new classes into the Java Agent Framework
- Can be used to augment the capabilities of a running Java Agent
- Is invoked when a manager request on a Managed Object Bean, or a locally initiated agent request (e.g. by another agent service) requests the creation of a new instance whose class is not loaded in the Agent Framework
- New class definition can be loaded from a remote class server
- Can be used to load new Managed Object Bean classes, or new Agent Service classes
- Example: A Java agent can be started without a logging service, and later, a logging service can be dynamically added when required
- To make this service work, the active security manager in the JVM hosting the Java agent must be configured to accept incoming libraries

(c) Copyright 1998 Subodh Bapat

Dynamic Class Loading Service

- The class loading service is itself an Agent Service, and so is an object that is created in the Java agent at start-up or at run-time
- The built-in Java class `java.lang.ClassLoader` can be used as the Java Agent's class loading service
- The class `java.lang.ClassLoader` defines the API for dynamic class loading: it is a standard Java class available in JDK
- Each Java agent may implement this API with its own customizations
- It is possible to have several instantiated class loader objects in the same Java agent, provided they are all registered with the service registration service
- Each class loader could use a different protocol for loading classes, and/or different class servers

(c) Copyright 1998 Subodh Bapat

Dynamic Native Library Loading Service

- Loads native (non-Java) libraries into a Java Agent Framework
- Is invoked when a new class that includes native code is loaded
- Purpose is to ease the development of Java Managed Object Beans and Native Libraries
- Can be used to augment the capabilities of a running Java Agent
- Can be loaded from a remote entity
- Can be loaded using the same mechanism as the Java Bean that calls the native library
- Smart agents can load libraries that are appropriate to the hardware platform and operating system on which they are running
  - the same Java Agent Framework can be smart enough to load different libraries depending on whether its operating environment is Solaris/Sparc, Solaris/Intel, Windows NT/Intel or Windows NT/Alpha

(c) Copyright 1998 Subodh Bapat

Dynamic Native Library Loading Service

- When the native library loading service is called, the Agent Framework implementation determines which class loader must be used (usually the same as the loader that was used to load the Java class that calls the native functions)
- If the class loader can also act as a native library loader, it can be used to load the native library
- If the class loader cannot act as a native library loader, then `java.lang.System` can be used to make a system call to load the native library
- Native library functions are called by Java code via the various *Java Native Interface* mechanisms
- The security manager in the JVM hosting the Java agent must be configured to accept incoming libraries
- Once a native library is loaded, the Java Agent is no longer 100% Pure Java, and is no longer portable

(c) Copyright 1998 Subodh Bapat

Example: Java Managed Object Bean

- This example shows a Java Managed Object Bean that represents the Ethernet interface `le0` of a Solaris system
- Assume that this Bean is a specific-mapped Bean from an SNMP MIB for the Ethernet interface
- For the sake of this example, this managed object has two attributes: `ifInPkts` and `ifOutPkts`; both are read-only
- Hence the Bean has two methods, `getIfInPkts()` and `getIfOutPkts()`, to obtain the values of these attributes
- There are no methods called `setIfInPkts()` and `setIfOutPkts()` since these attributes are read-only
- In this example, the Bean implements these functions by calling the native `kstat` (kernel statistics) UNIX system function from within the Java program

(c) Copyright 1998 Subodh Bapat

Example: Java Managed Object Bean

- Assume that `kstat` lives in the library `exampleLibraryPath/kstat`
- Define a Java class `KernelStat` that represents `kstat`
- Construct a `KernelStat` object by loading a native library in its constructor:

```
KernelStat (String modName, String instName) {
    moduleName = mod;
    instanceName = inst;
    AgentFramework.loadLibrary (this.getClass(),
           "exampleLibraryPath", "kstat");
}
```

(c) Copyright 1998 Subodh Bapat

Example: Java Managed Object Bean

- Define the Java Managed Object Bean that calls `KernelStat` as part of its instrumentation to obtain attribute values:

```
package ........kstat;
public class le0 implements java.io.Serializable {
    public le0() {
    ks = new KernelStat ("le", "le0");
    }
    public Integer getIfInPkts() {
    return (ks.getInteger ("ipackets"));
    }
    public Integer getIfOutPkts() {
    return (ks.getInteger ("opackets"));
    }
}
```

(c) Copyright 1998 Subodh Bapat

Java-based Agent Application Development

- Rapid development of agent applications can be achieved in a visual Bean development environment if all components of an agent are represented as *Java Beans:*
  - The AgentFramework Bean
  - The Service Beans (naming service, metadata service, event filtering/discrimination service, access control, event generation, persistence, relationship management, dynamic class loading, dynamic native library loading)
  - Managed Object Beans
- By connecting these Beans together in a visual Bean development environment, agent applications could be developed with minimal programming

(c) Copyright 1998 Subodh Bapat

Benefits of Java-Based Agents

- Software distribution and upgrade problem for agents in devices is drastically minimized
- Software distribution can be achieved with "agent pull" or "manager push" technologies
- New managed object class definitions can be dynamically added to the agent on the fly
- New intelligence and capabilities (services) can be dynamically added to the agent on the fly
- Configuration management and administration costs are lower

(c) Copyright 1998 Subodh Bapat

Products for Java Agent-Based Development

- *Java Dynamic Management Kit* (Sun Microsystems): for building network management agents
  - M-Beans represent Java managed objects
  - Service Beans represent Java agent services
  - Common Management Framework is a Java Agent Framework
- *Aglets Workbench* (IBM Tokyo Research Labs): general agents, not network management specific, but can be customized for network management
- *Voyager Agents* (ObjectSpace): general agents, not network management specific, but can be used for network management

(c) Copyright 1998 Subodh Bapat

Java in Network Management:

Java in the Manager

(c) Copyright 1998 Subodh Bapat

Benefits of Java-Based Manager Applications

- Enable rapid manager application development, using productivity benefits of Java
- Bring to the NOC the flexibility, portability and universality of Java
- Enable "lightweight management consoles"
- Add a universal user interface to high-end management platforms
- Enable new paradigms for configuration management

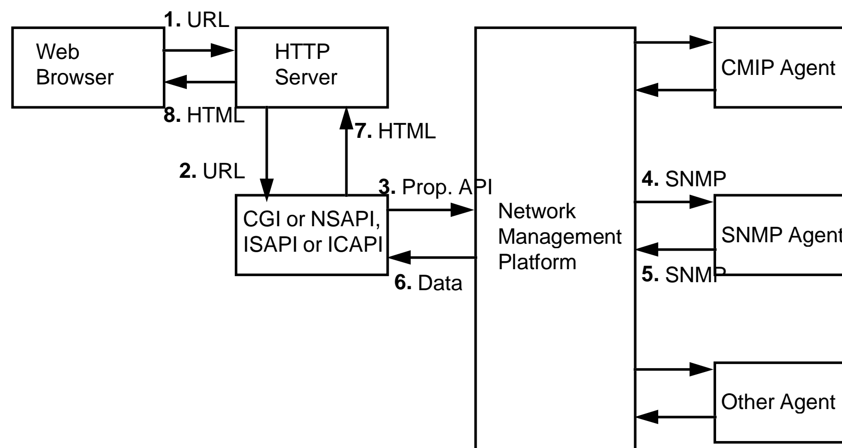
(c) Copyright 1998 Subodh Bapat

"Write Once, Manage Anywhere"

- *Browser-based management:* management applications can be written once, then executed on any JVM in any Web browser
- *Lightweight management console:* Any computer (e.g. a laptop) that runs a Web browser can become a management console
- *Empower field operations:* Operators and technicians can use their portable systems to dial in from anywhere, and have management console functionality
- *Integrated Operations, Support, and Testing:* All NMSs, NOCs and OMCs become accessible via a single browser-based interface, which can hyperlink between them

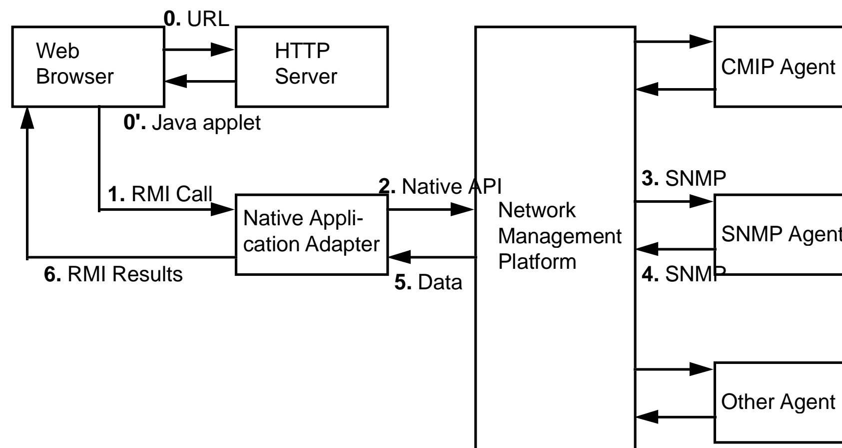
Legacy Web-based Management Approaches

- No Java used
- All management is dependent on CGI or Server APIs



Java Approaches to Web-based Management

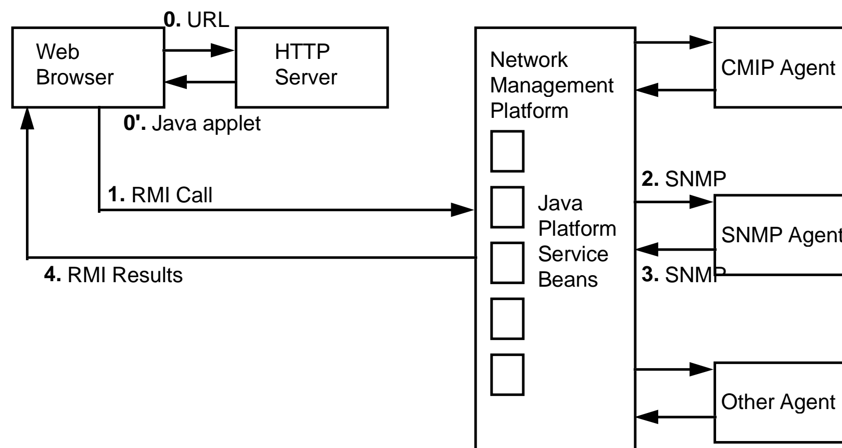
- More portable, platform-independent and HTTP-server independent



(c) Copyright 1998 Subodh Bapat

Java Management with Platform Service Beans

- Uses Java Service Beans in the Platform



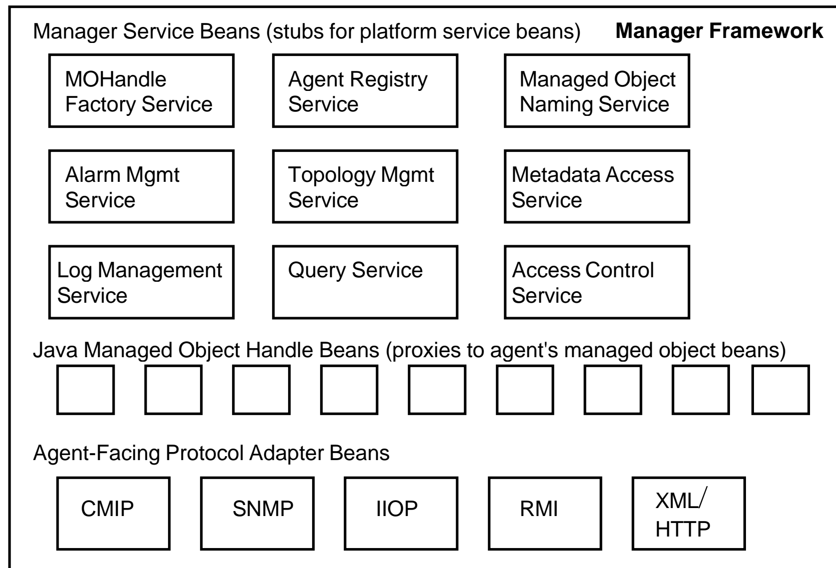
(c) Copyright 1998 Subodh Bapat

Architecture of a Java-Based Manager Application

- Java Virtual Machine
- The following Java Bean objects in a virtual machine:
  - A Java-based *Manager Framework*
  - Java *Managed Object Handles*
  - Access to a *Naming Service* to determine which agent hosts particular managed objects
  - Java-based SMI *metadata*
    - generic managed objects for non-Java SMI need a metadata database
    - metadata for managed objects defined in Java SMI, or specific-mapped managed objects from a non-Java SMI, is available via class reflection or Bean introspection

(c) Copyright 1998 Subodh Bapat

Architecture of a Java-Based Manager Framework



(c) Copyright 1998 Subodh Bapat

The Java Manager Framework

- Is a singleton class per Java Manager
- Provides a place to hook in manager services
- Provides a place to set application-wide defaults (e.g. application-wide default callbacks and other parameters)
- Provides a starting point to to interrogate the *Agent Naming Service* to resolve names of Java Managed Objects
- Provides a starting point for accessing SMI *metadata*
- Provides stubs to access Service Beans in the platform

(c) Copyright 1998 Subodh Bapat

Java Managed Object Handle Beans

- A Java *MOHandle* is a Java Bean in the space of a manager application, that represents a Java Managed Object Bean that lives in the agent
- A Java MOHandle proxies for the agent's managed object, i.e. a manager application executes operations on the real managed object by invoking methods on a MOHandle object that represents it
- A Java MOHandle is not necessarily the "stub" side of a managed object "skeleton"
  - the communication between a MOHandle Bean and the ManagedObject Bean need not necessarily be RMI
  - A MOHandle and a ManagedObject can communicate via any standard network management protocol

(c) Copyright 1998 Subodh Bapat

Java Managed Object Handle Beans

- If implemented as a Java Bean, the Java MOHandle can be visually manipulated in a Bean development environment
- The Java MOHandle Bean has methods to
  - determine its state
  - issue network management protocol requests to the agent (get/set/create/delete/action, etc.)
  - register to listen for events/traps emitted by the managed object
  - interrogate the metadata of the MOHandle's object class

(c) Copyright 1998 Subodh Bapat

Java Managed Object Handle Beans

- Can have a cache (locally stored managed object attribute values)
  - eliminates the need to visit the agent to read an attribute value each time
  - caching selectable on a per-attribute basis
  - values cannot be manipulated by application
- Can have a track list (attribute values to be automatically updated by the Bean implementation, based on events/traps received):
  - tracking selectable on a per-attribute basis
- Can have a stage (proposed attribute values):
  - values being prepared to be set on the managed object
  - will be used an arguments to a set request (or a create request)

(c) Copyright 1998 Subodh Bapat

Collections of Java Managed Object Handles

- A manager application can organize multiple MOHandle Beans in a collection
- The collection can be a standard Java container (`Set`, `List`, `Vector`, etc.)
- Operations on multiple managed object handles can be conveniently invoked by invoking a method on the whole collection
- Depending on the information model of the underlying managed object, this may translate into:
  - iterating over the individual MOHandles in the collection to issue a request on each one
  - issuing a single request to an agent hosting the Managed Object Beans, if the agent understands the concept of collections

(c) Copyright 1998 Subodh Bapat

Enumerated Collections of MOHandle Beans

- In an enumerated collection, the membership of the collection is fully controlled by the application
- An application may add or remove individual MOHandles at will
- Requests issued on the collections always translate into individual requests issued on individual MOHandle Beans in the collection

(c) Copyright 1998 Subodh Bapat

Rule-Based Collections of MOHandle Beans

- The membership of a rule-based collection is defined using a rule
- A snapshot of all ManagedObject Beans in all agents is taken to determine the membership of a rule-based collection
- The rule is defined using a Query Bean that is understood by a platform's Query service
- Information-model-independent Queries are:
  - a logical conjunction of attribute value predicates
- Information-model dependent Query Beans include, for example:
  - SNMP: a subnet mask
  - GDMO: scope and filter
  - CIM: transitive closure of an association

(c) Copyright 1998 Subodh Bapat

Java-based Manager Application Development

- Rapid development of manager applications can be achieved in a visual Bean development environment if all components of a manager are represented as *Java Beans:*
  - The ManagerFramework Bean
  - The stubs of Platform Service Beans that will be used by a manager application (naming service, metadata service, event filtering/discrimination service, access control, event generation, persistence, etc.)
  - MOHandle Beans and MOHandle Factory Beans
- By connecting these Beans together in a visual Bean development environment, manager applications could be developed with minimal programming

(c) Copyright 1998 Subodh Bapat

Benefits of Java Manager Application Development

- Manager applications run in a universal console using a familiar paradigm
- Easy, consistent extensibility and integration
- Builds upon existing management platforms, products, and protocols
- Based on open, industry standards
- Adds flexibility and scalability to create a true distributed management solution

(c) Copyright 1998 Subodh Bapat

Products for Java-Based Manager Development

- Sun Microsystems' *Solstice Enterprise Manager Java Supplement* has several APIs for writing Java applications to a platform
- Hitachi Telecom's Java implementation of the NMF TMN API
- Java API in WBEM SDK

(c) Copyright 1998 Subodh Bapat

Java in Network Management:

Java in the Platform

(c) Copyright 1998 Subodh Bapat

Benefits of Java-Based Management Platforms

- Implementations of network management platform services can be portable
- Service implementations can be upgraded easily
- Platform services can be interrogated for their metadata
- A standard mechanism for inter-service communication is available
- Java clients can orchestrate the co-ordination of multiple services across in a platform in a location independent-manner
- Services can be implemented as Enterprise JavaBeans

(c) Copyright 1998 Subodh Bapat

Network Management Platform Services

- A network management platform typically offers many services to its applications:
  - Message routing
  - Directory/Name resolution
  - Event distribution
  - Event logging and log management
  - Alarm management
  - Topology management
  - Queries and Reporting
  - Metadata access
  - Access Control
- All these services can be implemented as Beans in the platform
- Platform-based Beans ("server Beans") are called *Enterprise JavaBeans*

(c) Copyright 1998 Subodh Bapat

Enterprise JavaBeans

- Server-side component model for Java
- Enterprise JavaBeans typically have no visual representation at run-time (they may still have an icon to represent them visually at design time, so they can be manipulated in a visual application builder tool)
- Enterprise JavaBeans follow the same naming conventions as the JavaBeans specification for Bean properties, events, etc.
- Can be used to connect up with Manager Beans or Agent Beans to create network management applications with little or no programming

(c) Copyright 1998 Subodh Bapat

Enterprise JavaBeans

- Implementing platform services as Enterprise JavaBeans permits thin clients to take advantage of sophisticated server capabilities:
  - multithreading
  - multiprocessing
  - security and access control
  - pooling of resources
  - shared database access
  - concurrency control on database access
  - transactional support
  - replication and distribution of service components
- Componentizing platform services will permit the development of scalable, multi-tier management platforms

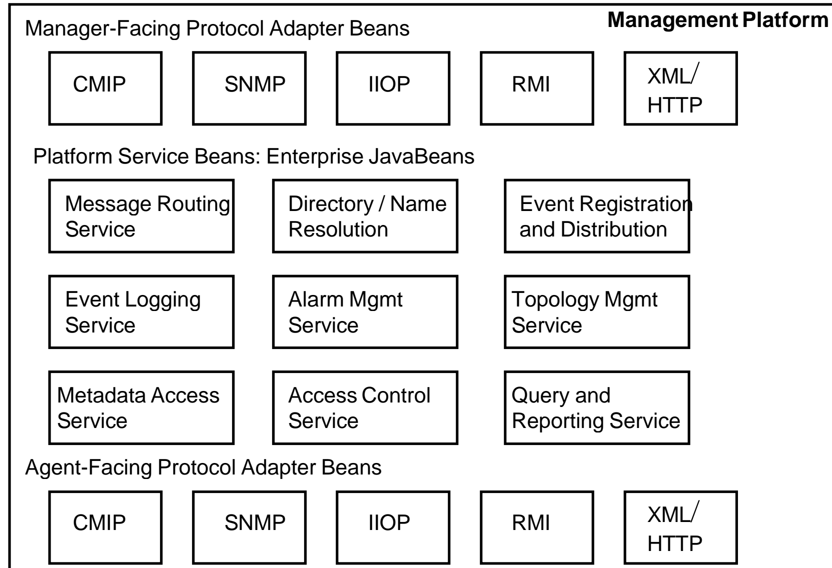
(c) Copyright 1998 Subodh Bapat

Enterprise JavaBeans

- Patterened after CORBA services and facilities
- May be considered the standardized Java API to CORBA services
  - for example, the JTS (Java Transactional Service) can be used as the Java API to CORBA OTS
- Can be used in a network management platform the same way distributed CORBA components can be used in a network management platform
- However, with Java, there is an added benefit of portability and dynamic class loading

(c) Copyright 1998 Subodh Bapat

Architecture of a Java-Based Platform



(c) Copyright 1998 Subodh Bapat

Platform Services as Enterprise JavaBeans

- Every service developer does not necessarily want to implement multithreading, concurrency control, resource-pooling, security, and transaction management separately in each component
- By using a component model, a service developer avails of the standardized and automated implementations of these features
- This enables the service developer to concentrate on developing the service logic
- Enables easy and rapid development of platform services
- For example, to customize transaction services, a service developer can define transaction policies while deploying the service Enterprise JavaBean by manipulating its properties
- Integrates with CORBA components via RMI/IIOP interworking

(c) Copyright 1998 Subodh Bapat

Enterprise JavaBeans to CORBA Mapping

- EJB/CORBA mappings are defined for 4 different areas:
  - Distribution Mapping: defines relationship between an EJB and a CORBA object (including mapping EJB Remote Interfaces to OMG IDL)
  - Naming Mapping: defines how COS Naming is used to locate EJB Container objects
  - Transaction Mapping: defines the mapping of EJB Transaction Support (JTS) to CORBA Object Transaction Service (OTS)
  - Security Mapping: defines the mapping of EJB Security to CORBA Security

(c) Copyright 1998 Subodh Bapat

EJB-CORBA Interoperability

- Current EJB specifications permit the following:
  - A CORBA client (in any language binding) can access an EJB deployed in a CORBA server
  - A client can mix calls to CORBA and EJB objects in a single transaction
  - A transaction can span multiple EJB objects that are located on multiple CORBA-based EJB servers, including servers from different vendors

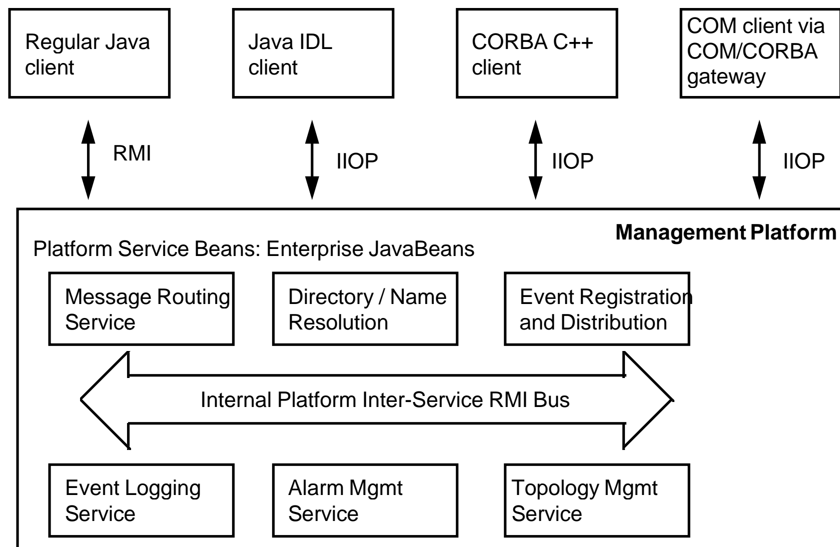
(c) Copyright 1998 Subodh Bapat

EJB-CORBA Interoperability

- Two kinds of clients are supported:
  - Regular EJB Java client:
    - Client uses Java APIs only (remote EJB interfaces)
    - Uses JNDI naming to locate the EJB
    - Is unaware of underlying use of IIOP
    - Makes regular RMI calls that go over RMI/IIOP
    - Client developer need define or generate no IDL (is done by tools)
  - Regular CORBA client:
    - Programmer must explicitly define IDL interfaces
    - Client written in any language that uses a language-specific binding to any stub generated by an IDL compiler
    - Uses COS Naming to locate the EJB
    - EJB server implemented as Java language bindings to IDL skeletons

(c) Copyright 1998 Subodh Bapat

Enterprise JavaBeans and CORBA



(c) Copyright 1998 Subodh Bapat

Alarm Management Service Bean

- Alarm Management Service can be implemented as an Enterprise JavaBean in a network management platform
- Alarm Management Service provides access to a client application to alarms that have accumulated in the platform
- Service may permit the creation of separate Alarm Log objects for different kinds of alarms, and log alarms to different AlarmLog objects depending on type
- Offers an API for client applications to acknowledge, clear, and delete alarms
- Offers no API for client applications to create or insert new alarms

(c) Copyright 1998 Subodh Bapat

Alarm Management Service Bean

- All methods below throw `AlarmLogException` and are `public` (not documented for succinctness):

```
public class AlarmLog implements EnterpriseBean {
  AlarmLog (String name);
  AlarmRecord[] getAlarms();
  AlarmRecord[] getAlarms
               (AlarmRecordId[] alarmRecordIds);
  AlarmRecord[] getAlarms (Query alarmQuery);
  AlarmRecord[] getAlarms (DeviceName deviceName);
  AlarmRecord[] getAlarms (TopologyNode topologyNode,
                           boolean getPropagated);
```

(c) Copyright 1998 Subodh Bapat

Alarm Management Service Bean (continued)

```
AlarmRecord[] getAlarms(int batchSize);
AlarmRecord[] getAlarms
                (AlarmRecordId[] alarmRecordIds,
                 int batchSize);
AlarmRecord[] getAlarms (Query alarmQuery,
                         int batchSize);
AlarmRecord[] getAlarms (DeviceName deviceName,
                         int batchSize);
AlarmRecord[] getAlarms (TopologyNode topologyNode,
                         int batchSize,
                         boolean getPropagated);
```

(c) Copyright 1998 Subodh Bapat

Alarm Management Service Bean

```
void performClearAlarms
                (AlarmRecordId[] alarmRecordIds);
void performClearAlarms ();
void performDeleteAlarms
                (AlarmRecordId[] alarmRecordIds);
void performDeleteAlarms ();
void performAcknowledgeAlarms
                (AlarmRecordId[] alarmRecordIds);
void performAcknowledgeAlarms ();
```

(c) Copyright 1998 Subodh Bapat

Alarm Management Service Bean (continued)

```
    void addAcknowledgeListener (AcknowledgeListener l);
    void removeAcknowledgeListener
                                (AcknowledgeListener l);
    void addClearListener (ClearListener l);
    void removeClearListener (ClearListener l);

    void addDeleteListener (DeleteListener l);
    void removeDeleteListener (DeleteListener l);
  } // end class AlarmLog
```


Alarm Bean

- All methods below throw `AlarmAttributeNotSetException` and are `public` (not documented for succinctness):

```
public class AlarmRecord implements EJBObject {

  AlarmRecordId getAlarmRecordId();
  String        toString();
  Date          getEventTime();
  Date          getLoggedTime();
  Date          getDisplayedTime();
  Date          getAcknowledgedTime();
  Date          getClearedTime();
  EventType     getEventType();
  MOName        getEmittingManagedObjectName();
  Severity      getSeverity();
  String        getProbableCause();
```

Alarm Bean (continued)

```
boolean       isAcknowledged();
void          performAcknowledge();
String        getAcknowledgingOperator();
String        getAcknowledgementText();

boolean       isCleared();
void          performClear();
String        getClearingOperator();
String        getClearingText();

void          performDelete();
```

(c) Copyright 1998 Subodh Bapat

Alarm Bean (continued)

```
  void addAcknowledgeListener (AcknowledgeListener l);
  void removeAcknowledgeListener
                              (AcknowledgeListener l);
  void addClearListener (ClearListener l);
  void removeClearListener (ClearListener l);

  void addDeleteListener (DeleteListener l);
  void removeDeleteListener (DeleteListener l);

} // end class AlarmRecord
```

- Still need to define interfaces for clients to implement:

```
public interface ClearListener {...};
public interface AcknowledgeListener {...};
public interface DeleteListener {...};
```

(c) Copyright 1998 Subodh Bapat

Topology Management Service Bean

- A topology service in a management platform provides an abstract notion of the topology of the network
- The topology service maintains information about the state of the physical devices and their connections as *abstract topological graphs* (nodes and relationships)
- Nodes in a topology service need not necessarily correspond to physical devices; could be logical entities (such as a site, building, campus, or city)
- Topology Management Service Bean could be implemented as the Enterprise JavaBean API to the CORBA Topology Service

(c) Copyright 1998 Subodh Bapat

Topology Management Service Bean

- Basic topology service bean could export the following interfaces:
  - node creation and deletion
  - relationship management (graph maintenance)
  - node containment
- Advanced topology service bean interfaces may include:
  - integration with maps, GIS, and urban facilities layouts
  - layout optimization
  - propagation of state among topology nodes
    - "sideways" propagation of alarm conditions
    - "upward" propagation of alarm counts
    - "upward" propagation of alarm severities

(c) Copyright 1998 Subodh Bapat

Polling Service Bean

- A polling service in the platform permits the polling of managed objects that cannot issue traps/notifications
- For scalability, polling should not be driven by a manager application, but should occur via a service Bean that executes "near the bottom end" of the platform
- The Polling Service Bean can accept poll requests from managers, commence polling on the "agent-facing" side via a standard management protocol, and fire Bean events on the "manager-facing" side when specified thresholds are exceeded

(c) Copyright 1998 Subodh Bapat

Polling Service Bean

- The Polling Service Bean can be an Enterprise JavaBean that exports remote Bean interfaces for:
  - manager applications to create a `PollRequest` EJB object with a particular id
  - define the agents and/or managed objects to be polled
  - specify the polled attributes and polling frequency
  - specify the threshold values which will trigger the firing of Bean events
  - register listener objects on which methods can be invoked when the threshold conditions are met

(c) Copyright 1998 Subodh Bapat

Enterprise JavaBeans for Other Platform Services

- Similar EJB interfaces can be defined for other platform services:
  - Access Control
  - Metadata Access
  - Log Management
  - Event Registration and Filtering
  - Event Correlation
  - Queries
  - Discovery

(c) Copyright 1998 Subodh Bapat

Java in Network Management:

Network Management APIs in Java

(c) Copyright 1998 Subodh Bapat

Network Management APIs in Java

- **Protocol-level APIs (low-level APIs)**
- Managed Object-level APIs (high-level APIs)

(c) Copyright 1998 Subodh Bapat

Protocol-level APIs (low-level APIs)

- Direct APIs to a management protocol stack
- Application needs to be aware of details of protocol PDUs
- API classes represent structures of particular PDUs in the protocol:
  - `SnmpSetRequestPdu`
  - `SnmpGetResponsePdu`
  - `CmipMCreateRequestPdu`
  - `CmipMDeleteConfirmationPdu`
- Management applications load up an instance of such a class with data, and then give it to the appropriate Protocol Service Bean to issue out of the VM

(c) Copyright 1998 Subodh Bapat

Protocol-level APIs (low-level APIs)

- APIs exist for many common protocol stacks:
  - SNMP
  - CMIP
  - TL-1
  - proprietary
  - other general protocols that can be used to carry management messages e.g. RMI, XML/HTTP, IIOP
- Depending on how and where the protocol-level API will be used, it may have different profiles:
  - agent-facing
  - manager-facing

(c) Copyright 1998 Subodh Bapat

Examples of Protocol-level APIs in Java: SNMP

- Advent SNMP Stack

```
public class SnmpPDU {
    public SnmpPDU (SnmpAPI api);
    public void addNull (SnmpOID oid);
    public void fix();
    public long round_trip_delay ();
    public String printVarBinds();
    public int get_encoded_length();
    public boolean decode() throws SnmpException;
    public SnmpPDU copy();
}
```

(c) Copyright 1998 Subodh Bapat

Examples of Protocol-level APIs in Java: CMIP

- Can be used to rapidly prototype browser-based managers and agents against a CMIP stack
- Very handy for writing quick testing applications
- Examples: Any Javatized version of the CMIS/C++ component of the NMF's TMN/C++ API standard:
  - Presents a protocol-level API in Java to CMISE service primitives (M-GET, M-SET, etc.)
  - Different Java classes for each PDU variation (request, indication, response, confirmation)
  - Java classes for CMISE parameters (Attribute, AttributeId, Scope, Filter, etc.)
- Products implementing this include:
  - Sun's TMNscript Java Client/Server Gateway
  - UH Communications Q3ADE CMIS API for Java

(c) Copyright 1998 Subodh Bapat

Management Protocol Adapter Beans

- Protocol Adapter Beans use the Protocol Service Beans to adapt a standard network management protocol to methods on Java Managed Object Beans
- Management Protocol Adapter Beans convert management protocol requests (from particular management protocols) to native Java calls on Java Managed Object Beans
- They also convert notifications emitted by Java Managed Object Beans into the appropriate event formats of the management protocols

(c) Copyright 1998 Subodh Bapat

Management Protocol Adapter Beans

- Just like there are two kinds of Protocol Service Beans, there are two kinds of Management Protocol Adapter Beans, depending on their capabilities:
  - *Manager-facing Protocol Adapter Beans*, and
  - *Agent-facing Protocol Adapter Beans*
- A Management Protocol Adapter Bean can support both an agent-facing interface and a manager-facing interface, if it is used (for example) in a mid-level manager

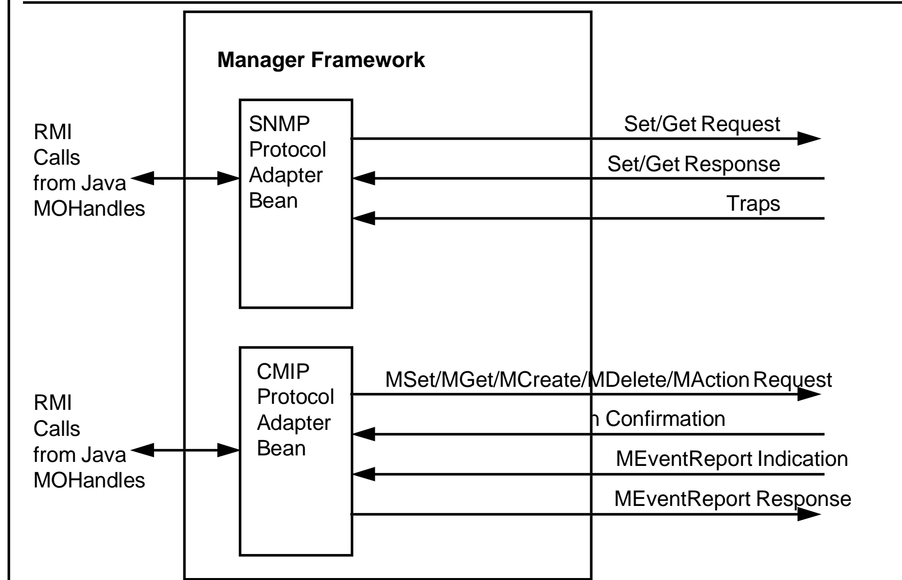
(c) Copyright 1998 Subodh Bapat

Agent-Facing Protocol Adapter Beans

- Agent-facing Protocol Adapters live in a Java Manager Framework, and have the ability to
  - send requests to agents
  - receive responses from agents
  - receive notifications from agents
  - send responses to notifications from agents (if the management protocol so allows)

(c) Copyright 1998 Subodh Bapat

Agent-Facing Protocol Adapter Beans

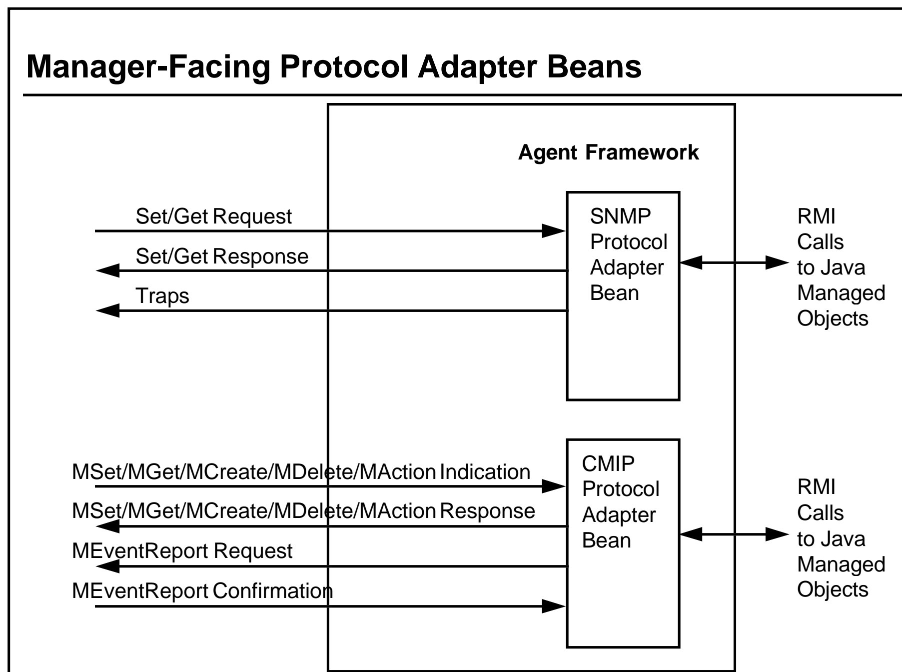


(c) Copyright 1998 Subodh Bapat

Manager-Facing Protocol Adapter Beans

- Manager-facing Protocol Adapters live in a Java Agent Framework, and have the ability to
  - receive manager requests
  - send responses to manager requests
  - send notifications to managers
  - receive responses from managers to notifications (if the management protocol so allows)

Manager-Facing Protocol Adapter Beans



Network Management APIs in Java

- Protocol-level APIs (low-level APIs)
- **Managed Object-level APIs (high-level APIs)**

(c) Copyright 1998 Subodh Bapat

Managed Object APIs

- In recent years the emphasis has shifted away from protocol battles
- Application developers no longer like to write applications to low-level protocol-level APIs; don't want to know PDU content and format
- Application developers would like to think of the world as distributed objects
- Network management is nothing but an example of communication between distributed objects
- In network management, future applications will be written to manipulate managed objects by invoking local and remote methods on them; actual protocols used to convey those operations will be unimportant
- Therefore, object-oriented information models (Java, CORBA, GDMO, CIM) become very important
- APIs to information models become very important.

(c) Copyright 1998 Subodh Bapat

Managed Object APIs

- *Managed Object APIs* are high-level APIs in Java that permit an application to deal with managed object abstractions, without worrying about the details of the network management protocol being used
- Multiple dimensions for consideration, depending on how and where the high-level API will be used:
  - information model-aware/information model-independent
  - manager-side/agent-side
  - generic/specific

(c) Copyright 1998 Subodh Bapat

Management Information Models

- Many different ways to model the information that is of interest to management *(Structures of Management Information)*:
  - IETF SMI (managed objects defined in SNMP MIBs)
  - ITU MIM/SMI (managed objects defined in GDMO documents)
  - DMTF SMI (managed objects defined in CIM schemas)
  - Java Notation itself (managed objects defined in source Java)
- Aimed at different markets:
  - SNMP SMI: Intranet and ISP equipment management
  - GDMO: Telecommunications equipment and service provider management
  - CIM: Systems and Applications management
  - Java Notation: low-end devices, embedded systems

(c) Copyright 1998 Subodh Bapat

Management Information Models

- History of evolution of information models has many lessons to offer
- Each information modeling technique starts off with the goal of being the unifying paradigm
- *Unifying paradigms have never worked!*
- Over time, each information model becomes associated with a particular protocol that manipulates the constructs of that information model
- Over time, mappings are developed to translate models from one SMI syntax to another; these sometimes work, sort of, kind of

(c) Copyright 1998 Subodh Bapat

Information Model APIs: Conflicting Requirements

- *Versatility, Sophistication, and Semantic Richness:* Developers who develop applications to a particular information model would like a Java API that fully expresses the semantics of that information model, and gives them full control over its capabilities
- *Ease of Application Development:* Developers find learning information models hard, want to be able to write management applications without learning any modeling syntax
- How do we satisfy both kinds of developers?

(c) Copyright 1998 Subodh Bapat

Information Model Independent APIs

- A class of applications can be developed which are insensitive to the information model of the underlying managed object
- Any application that interacts with the platform using only service-specific Enterprise JavaBean APIs need not use the Managed Object APIs at all
- Examples:
  - Some alarm display applications may not care whether the device that generated the alarm is managed using an SNMP agent or a CMIP agent
  - A provider's equipment asset display application may not care whether the topology nodes are managed using an SNMP agent or a CMIP agent
- Such applications may need to interact directly with the managed object only minimally, and simple generic get and set methods will suffice

(c) Copyright 1998 Subodh Bapat

Manager Application Beans: MOHandle

- A Java *MOHandle* is a Java Bean in the space of a manager application, that represents a Java Managed Object Bean that lives in the agent
- A Java MOHandle proxies for the agent's managed object, i.e. a manager application executes operations on the real managed object by invoking methods on a MOHandle object that represents it
- A Java MOHandle is not necessarily the "stub" side of a managed object "skeleton"
  - the communication between a MOHandle Bean and the ManagedObject Bean need not necessarily be RMI
  - A MOHandle and a ManagedObject can communicate via any standard network management protocol

(c) Copyright 1998 Subodh Bapat

Manager Application Beans: MOHandle

- A MOHandle does not implement or execute any managed object behavior
- All managed object behavior executes only in a real managed object, which must always live in an agent
- Behavior may be informally defined or may be formally defined in Java code
- A manager application can cause the real managed object to execute its behavior by invoking methods on the MOHandle, which will transparently send it network management protocol messages

(c) Copyright 1998 Subodh Bapat

Information Model Independent MOHandle

- An information model independent MOHandle can represent a managed object defined in any information model
  - Can represent an instance of a GDMO managed object class
  - Can represent an instance of a CIM managed object class
  - Can represent an instance of an SNMP "managed object class" (where object groups are mapped into classes using any standard mechanism that makes SNMP MIBs object-oriented, e.g. one similar to the IIMC mapping rules)
  - Can represent an instance of a managed object defined directly in Java notation
- Abstracts the common features of all information models into an information model independent MOHandle
- Is useful for a class of applications that only want to do basic things with a managed object

(c) Copyright 1998 Subodh Bapat

Information Model Independent MOHandle

- The implementation of the MOHandle object in any system knows what information model its managed object is specified in (since it must send it the right kind of protocol message)
- The MOHandle interface, however, need not expose this knowledge
- The MOHandle interface need only include:
  - generic get and set methods:
    - getAttribute(String attributeLabel)
    - setAttribute(String attributeLabel, AnyValue anyValue)
  - event registration methods for listening to changes in the managed object

(c) Copyright 1998 Subodh Bapat

Information Model Independent MOHandles

- Useful things can be done with information model independent MOHandles:
  - Attribute caching: provide a list of attributes to cache locally in the MOHandle so every frequently used values can be retrieved with a "local get", saving network traffic
  - Attribute tracking: provide a list of attributes whose cached values can be tracked dynamically by the MOHandle implementation based on notifications/traps received from the network
  - Attribute staging: provide a place to prepare proposed attribute values before sending them down the real managed object in the agent

(c) Copyright 1998 Subodh Bapat

Collections of Java Managed Object Handles

- A manager application can organize multiple MOHandle Beans in a collection
- The collection can be a standard Java container (`Set`, `List`, `Vector`, etc.)
- Operations on multiple managed object handles can be conveniently invoked by invoking a method on the whole collection
- Depending on the information model of the underlying managed object, this may translate into:
  - iterating over the individual MOHandles in the collection to issue a request on each one
  - issuing a single request to an agent hosting the Managed Object Beans, if the agent understands the concept of collections

(c) Copyright 1998 Subodh Bapat

Enumerated Collections of MOHandle Beans

- In an enumerated collection, the membership of the collection is fully controlled by the application
- An application may add or remove individual MOHandles at will
- Requests issued on the collections always translate into individual requests issued on individual MOHandle Beans in the collection

(c) Copyright 1998 Subodh Bapat

Rule-Based Collections of MOHandle Beans

- The membership of a rule-based collection is defined using a rule
- A snapshot of all ManagedObject Beans in all agents is taken to determine the membership of a rule-based collection
- The rule is defined using a Query Bean that is understood by a platform's Query service
- Information-model-independent Queries are:
  - a logical conjunction of attribute value predicates
- Information-model dependent Query Beans include, for example:
  - SNMP: a subnet mask
  - GDMO: scope and filter
  - CIM: transitive closure of an association

(c) Copyright 1998 Subodh Bapat

Information Model Aware Subclasses

- For management application developers who need the full power of an information model, information model aware subclasses of MOHandle can be defined:
  - CIM MOHandle: is aware of CIM constructs (e.g. associations) and protocol operations
  - GDMO MOHandle: is aware of GDMO constructs (e.g. name bindings) and protocol operations
  - SNMP MOHandle: consists of a "managed object mapping" of SNMP MIBs (e.g. turning object groups into object classes using some standard mapping
- These are still generic, i.e. even though information model aware, these Beans can represent any managed object of the appropriate information model
- Useful for writing information-model-aware applications

(c) Copyright 1998 Subodh Bapat

Specific MOHandle Beans

- Specific MOHandle Beans are generated by a compiler. e.g.

  CIM `printer.mof` file `-->` compiler `-->` `printer.java` class
  SNMP `hub.mib` file `-->` compiler `-->` `hub.java` class
  GDMO `switch.gdmo` file `-->` compiler `-->` `switch.java` class

- These provide type-safe access to particular attributes, methods, etc. of managed objects
- Are always information model aware
- Can be subclassed from generic Beans so that all the interfaces of generic Beans are available (e.g. ability to cache and stage attributes, ability to be put in containers, etc.)
- Each generated `.java` file can then be further compiled through `rmic` to generate stubs and skeletons for distribution
- Useful for writing type-safe information-model-aware applications

Metadata Bean APIs

- Each MOHandle Bean can be queried for its metadata object
- Metadata Beans returned are always information-model aware:
  - GDMO Managed Object Class Template
  - CIM Class Schema
  - SNMP MIB Object Types and Groups
- Metadata Beans can then be queried in a structured manner for detailed metadata constructs

Metadata Bean APIs

- For specific MOHandles, which have type-safe methods generated by a compiler from a source information model definition, much metadata information may be simply available from Bean introspection
- Thus, metadata can be visually inspected in a Bean-based application development environment
- For generic MOHandles, no information-model metadata is available via Bean introspection
- Metadata access must be done via a metadata database:
  - metadata database must be accessible at run time to do validation of method invocations on the MOHandle Bean against metadata
  - metadata database must be accessible at design time to be able to build applications visually by querying against Bean metadata

(c) Copyright 1998 Subodh Bapat

Metadata Bean APIs

- How can metadata information for generic MOHandles be made available via Bean introspection?
- Options are:
  - Replace the `class` class which is returned by the `getClass()` method in Java reflection. Provide your own implementation of the `class` class which actually reads metadata off a metadata database, rather than from the Java .class file.
  - Replace the Bean `Introspector`. Provide your own implementation of `Introspector` which reads metadata off a metadata database.
  - Provide your own implementation of `BeanInfo` which reads metadata off a metadata database.

(c) Copyright 1998 Subodh Bapat

Products: Network Management APIs in Java

- Sun Microsystems' *Solstice Enterprise Manager Java Supplement* has several APIs for writing Java applications to a platform
- Hitachi Telecom's Java implementation of the NMF TMN API
- Java API in WBEM SDK

(c) Copyright 1998 Subodh Bapat