Framework for managing large scale component-based distributed applications using JMX

Julio Guijarro, HP Labs. Bristol, UK. julio_guijarro@hp.com
Manuel Monjo, Universidad Politécnica de Madrid, Spain. monjovillalba@yahoo.es
Patrick Goldsack, HP Labs. Bristol, UK. patrick_goldsack@hp.com

May 2002

Abstract

This paper presents a extended Java Management eXtensions service developed for the SmartFrog framework to provide a standard instrumentation layer for component-based distributed systems. This extended JMX service, when used in combination with SmartFrog, pursues several goals:

- Extend JMX to be able to manage large-scale distributed applications.
- Configuration management and deployment of complex JMX installations.
- Transparent integration of a JMX instrumentation layer with SmartFrog applications and SmartFrog infrastructure.
- Easy description and deployment of MBeans in a distributed environment.

Keywords: distributed, large-scale, configuration, SmartFrog, JMX, management, instrumentation.

1 Introduction and Motivation

The exponential growth in the scale and complexity of the data centre of the future, referenced also as “computing utility” or “Grid computing”, means that the current manual controls and procedures are becoming unsustainable.

In HP Labs we are working on aspects of the programming model for the computing utility. Our research includes how to specify what we want the utility to do, then how to turn that specification into a running system deployed securely and reliably on the resources of the utility, and managed through its lifecycle.

As part of our research, we are developing "SmartFrog" [7, 8] (or SF) -- a framework for specifying system configurations, and deploying and managing complex collections of components across distributed resources.

Java Management Extensions (JMX) is an emerging specification to instrument disparate pieces of Java code. The JMX architecture can cope with the integration of different management technologies (SNMP, CIM, TMN …). However, JMX does not yet addresses a management model for large-scale distributed applications running in dozens of servers.

This paper describes the JMX service developed for the SmartFrog framework. First of all, it introduces the SmartFrog framework, explaining briefly its architecture. Then, it gives an overview of JMX and the limitations of its present specification to manage large-scale applications. Next, it explains the changes introduced to JMX and its integration with the SmartFrog framework. Finally, the main ideas are summarised in the conclusions.

2 SmartFrog as a Distributed Application Framework

SmartFrog is a framework for the development of configuration-driven systems. It was originally designed as a framework for building and managing large monitoring systems where flexible configurations are essential.

The name reflects its basic design conception – the Smart Framework for Object Groups. It defines systems and sub-systems as collections of software components with certain properties. The framework provides mechanisms for describing these component collections, deploying and instantiating them and then managing them during their entire lifecycle.

The framework consists of four major aspects (figure 1):

- The SmartFrog configuration description environment consisting of a description notation and tools to enable the storage, validation and manipulation of these descriptions.
- The SmartFrog component model defining the interfaces that a software component (or a management adapter for a component) should implement. These interfaces are to support the various lifecycle operations such as creation, versioning and termination, as well as management actions such as accessing status information
- The SmartFrog configuration management system which uses these descriptions and management adapters to instantiate the software components and to monitor them throughout their lifecycle in a secure way, including an integrated run-time environment providing capabilities such as discovery and naming.
- Off-the-shelf services. SmartFrog comes with a number of useful components, applications and enhancements. These services are available to their immediate use. Some of these services are: a Java Management eXtension service (JMX) that enables dynamic management and monitoring of the deployed applications, SLP discovery, workflow description, partition membership protocol and scripting.

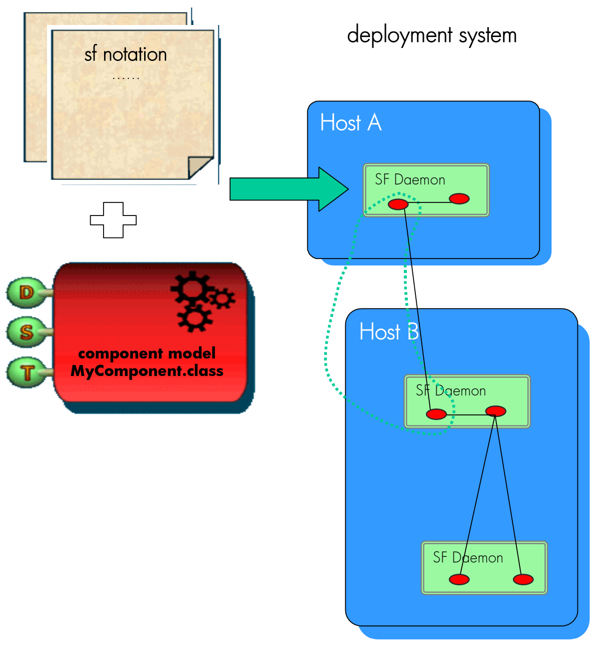


Figure 1. SmartFrog Framework Architecture

2.1 Notation

The SmartFrog ‘notation’ is in fact defined as a set of open data structures. In principle, this definition can support a number of parsers that provide textual versions of the notation (for example using XML as a surface syntax).

The notation is object-oriented, supporting inheritance and extension of configuration descriptions. These descriptions consist of component definitions, associations and relationships between the components, and workflows associated with the lifecycle of the components and the system as a whole. The descriptions may be parameterized enabling multiple instantiation with different configuration data, and validations may be provided which verify that these instances are correct before an attempt is made to instantiate the configuration.

The current version of SmartFrog, though in principle able to support multiple textual languages, only provides a specialized notation “out-of-the-box”. Others are in preparation for future releases.

The notation is not used to define behaviour, merely the structure of collections of components and their relationships with other collections. It is not a programming language. The behavioural part of a component is assumed to be defined in an existing programming language (such as C or Java) and the component will be started as appropriate by the SmartFrog configuration management system. Currently only Java is tightly integrated. Java adaptors must be used to wrap code written in other languages; these are relatively simple to implement.

An important and successful part of the original SmartFrog system was the design of the configuration description language. The language introduced a number of important features that have proved essential. In particular:

- the use of hierarchical sets of attributes (name/value pairs) as the basis of the description, providing a naming scheme for attributes
- the use of prototyping as the basis for creating new descriptions from old
- the use of the linking mechanism, associating one attribute with the value of another

2.2 Components

The component model supported by SmartFrog is a simple, extensible set of interfaces providing access to key management actions – such as instance creation, configuration, termination and so on. A component may be fully integrated (i.e. it may implement the defined management interfaces directly, and hence be written in Java) or it may be independent in which case a management adapter must be provided. Several standard management adapters or base integrated components have been written to provide common behaviours and these may be extended or modified as appropriate.

Each component (or adapter) must implement a published lifecycle, implemented as a set of action routines that the environment invokes in the appropriate order and at the right time to carry out the configuration or other management task required. The lifecycle process is governed and controlled by the definition of workflows within the SmartFrog system to provide a very flexible and adaptable environment for carrying out the various configuration tasks.

A complete set of APIs is available to the components that allow them to access the configuration information, locate other components as defined in the configuration and to alter the running configuration if so desired.

Besides all this, the framework also provides users with the ability to alter the low-level semantics by replacing functional units, yet providing standard capabilities by offering default implementations of these units.

2.3 Environment

The SmartFrog configuration and management infrastructure is supported by a collection of services, such as

- deployment – the distribution of code, configuration data and instantiating the components in the right place with certain ‘transactional’ guarantees;

- discovery and naming – providing a number of binding services to allow components to locate each other and communicate;
- management – each and every component is manageable though the provided services with no developer effort.

These services are incorporated so as to provide a seamless and coherent programming and configuration model. The benefits of this approach are those of providing configuration abstractions to component developers that allow multiple configurations of different scale to be produced without altering the components in any way. The environment is broken into several well-defined functional units, each of which has some specific role to play. Furthermore each of these operates through some well-defined and open interfaces, so it easy to replace the existing functional units, or even to make the selection of which instance a functional unit to use part of the configuration description.

The SmartFrog system supports the notion of a binding between software components and provides multiple ways – determined by the environment and driven by the configuration descriptions – for these bindings to be resolved. The precise binding mechanism is handled by the SmartFrog environment as defined by the configuration and it is transparent to the programmer.

3 JMX Overview

Java Management eXtensions [1, 2], or JMX, provides a reusable framework for exposing applications to remote or local management tools.

The JMX specification distinguishes four different levels (Figure 2) inside the JMX architecture, of which only the first two levels are covered by the specification:

- Instrumentation level: provides a specification for implementing JMX manageable resources. The instrumentation of a given resource is provided by one or more Managed Beans or MBeans. These MBeans are managed through the Agent level.
- Agent level: manages the resources and makes them available to remote management applications. A JMX Agent consists of an MBean Server and a set of services for handling MBeans and providing added value to the system. The Agent level decouples the management applications from the resources and it can be configured dynamically to meet the needs of the management system by registering services to the MBean server.
- Distributed services level: defines interfaces and components for implementing JMX managers, allowing management applications to interact with agents, providing views of the Agent and its MBeans by mapping their semantics to other protocol (SNMP, HTML, etc.) and creating logical views relevant to the end user. The current

specification leaves the definition of these interfaces and other functionality for a future version[1].

- Additional Management Protocol APIs. The MBean server does not implement remote access methods. The remoteness in the agent level is achieved by configuring the JMX agent with one or more JMX connector or protocol adaptor MBeans. The connectors or protocol adaptors allow the management applications to access a remote JMX agent via specific protocols such as SNMP, CIM/WBEM, etc [2].

Since the MBean server and registry is defined by the JMX specification [1], any compliant MBean is able to plug into any compliant MBean server.

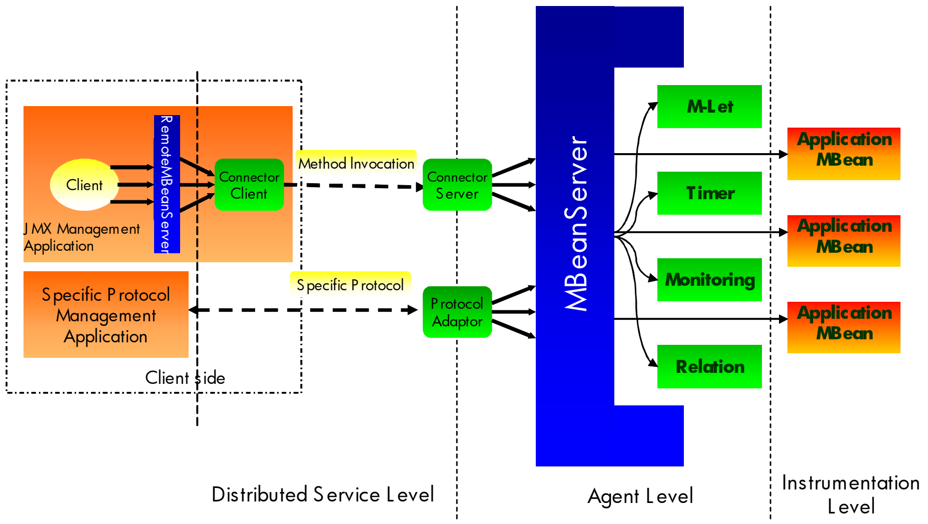


Figure 2. JMX Reference Architecture

3.1 *Limitations of the JMX v1.0 Specification in Large-Scale Systems*

Some of the limitations we can find in the current JMX specification have to be more with the lack of maturity of this technology rather than other defects and its initial intention of agents managing local resources.

- The Reference Implementation (and almost all other implementations found by the authors) considers that an Agent and its components, including MBeans will be

[1] The Java Specification Request 160 [7] (or JMX 1.5) will define the distributed services level in greater detail. Among other things, JSR 160 will define connectors to access remote MBeanServers, and, possibly, discovery mechanisms for them.

instantiated in the same Java Virtual Machine. In a pure distributed system it is desirable that the JMX Agent could be distributed as well. This is a very common problem when used to instrument distributed systems, as shown in [9].

- JMX in its present specification wouldn't be able to cope with thousands of MBeans deployed in a distributed fashion among hundreds of machines, as required in large-scale distributed applications.
- The instantiation of a complex JMX infrastructure has not yet been considered.
- There is no definition of how to contact or discover already instantiated JMX Agents, so that other Agents or external management tools could contact them.
- A specific limitation for SmartFrog framework is the flat nature of the data retrieved through JMX, which does not fit very well into the hierarchical nature of the SmartFrog data structures. A mapping mechanism is needed to resolve this problem if we want to have total control of the SF infrastructure.

The JSR 160 [6] will define the distributed services level. JSR 160 at the time of writing this paper (May 2002) is in its early stages. This specification is expected to provide answers to some of these problems.

3.2 Requirements for Integrating JMX with SmartFrog

The above limitations basically define the requirements for the extended JMX service implemented for the SF framework that can be summarised as:

- Ease of use: exposing an application to be managed should be as transparent as possible for the application designer.
- Scalability: management infrastructure should be able to grow according to our needs without impacting performance and ease of management. It should scale to manage tens of thousands of components in hundreds of different locations.
- Flexibility: should be fully configurable to fit the management needs of the applications.

4 sfJMX, an Extended JMX Service for Fully Distributed Systems

The initial motivation for integrating a JMX service in the SF framework (henceforth sfJMX) is to be able to provide a standard way of exposing to external management tools, such as HP Open View, the management of the SF infrastructure itself and each of the deployed applications. This has to be done in the least intrusive way possible. At the same time, it is desired to provide SmartFrog with the means to manage and deploy standard JMX-enabled resources that do not necessary need to be SF aware. This should allow any SF application to interact with management protocols that can interoperate with JMX such as SNMP or CIM/WBEM.

The sfJMX service gives distributed applications access to a JMX based infrastructure to both expose their manageable resources and to take advantage of all the services and capabilities that JMX offers.

Another advantage of the combined use of SF and JMX is that SF can be used as the basic infrastructure to describe, configure, deploy and manage the JMX infrastructure. This simplifies the instantiation of complex JMX installations in a fully distributed fashion. The SmartFrog notation is used to completely describe the desired installation; everything can be described from the metadata needed for each MBean to the connections between the different elements like MBean to Agent or Agent-to-Agent. The use of the SF notation simplifies to a great degree the description and creation of JMX metadata for MBeans through the use of its object-oriented features and support for inheritance. In most cases, this metadata is automatically created or templates are provided to cover most generic cases.

The integration of JMX with SF applications is simple and clean. The application designer doesn't need to include anything special to expose the management information as defined by the JMX specification. To add JMX management to a SF application it is only necessary to pass a reference to the application component whose interface has to be exposed to a special SF component. This SF component, named MBeanDeployer, will create the metadata required by JMX and register the appropriate MBean(s) in the desired JMX Agent on behalf of the application component(s).

The manageable information (MBeans) can be distributed throughout a network without caring about where the real manageable resource is located. In the same way, applications can use any service located in any point of the network. These services, according to the JMX architecture, are mainly *pluggable* MBeans registered in different JMX Agents.

The MBeanDeployer is the SF component responsible for locating the specified JMX Agent, create, if necessary, and registering the required MBeans on behalf of components of the distributed application. This model relieves the application of including code for exposing their manageable resources.

Using the SF notation it is also possible to configure and deploy a JMX infrastructure. The description contains information about the configuration of the JMX Agents and the different bindings amongst them. It is possible to create hierarchies of Agents with different levels of granularity. The network of JMX Agents can be as complex as needed having the possibility of deploying many kinds of topology depending on the requirements of the applications.

A possible complex installation is shown in Figure 3. Also in the appendix there is a complete example with its descriptions.

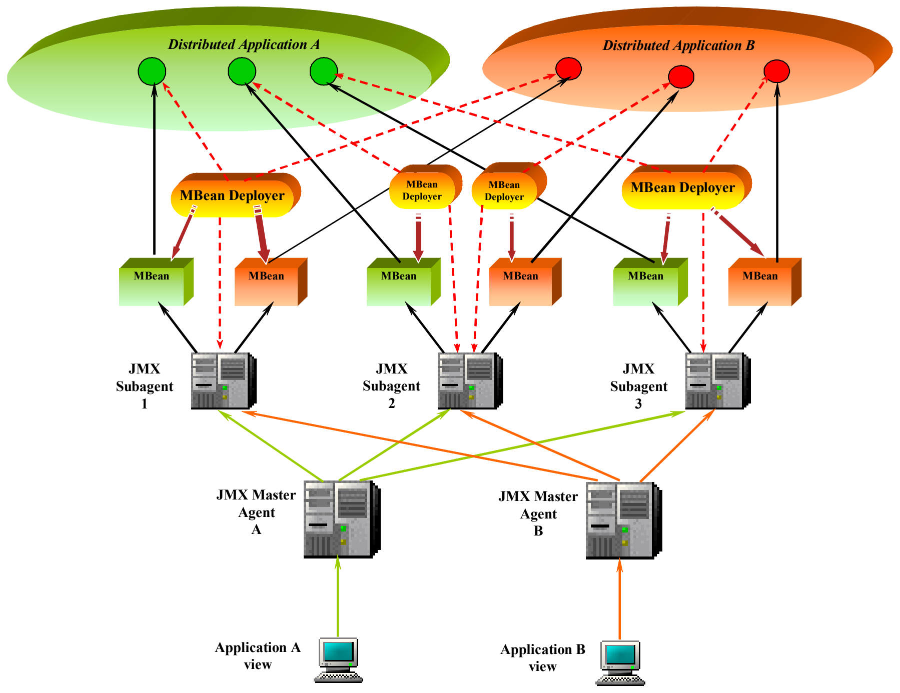


Figure 3. Example of a Complex Distributed JMX Topology

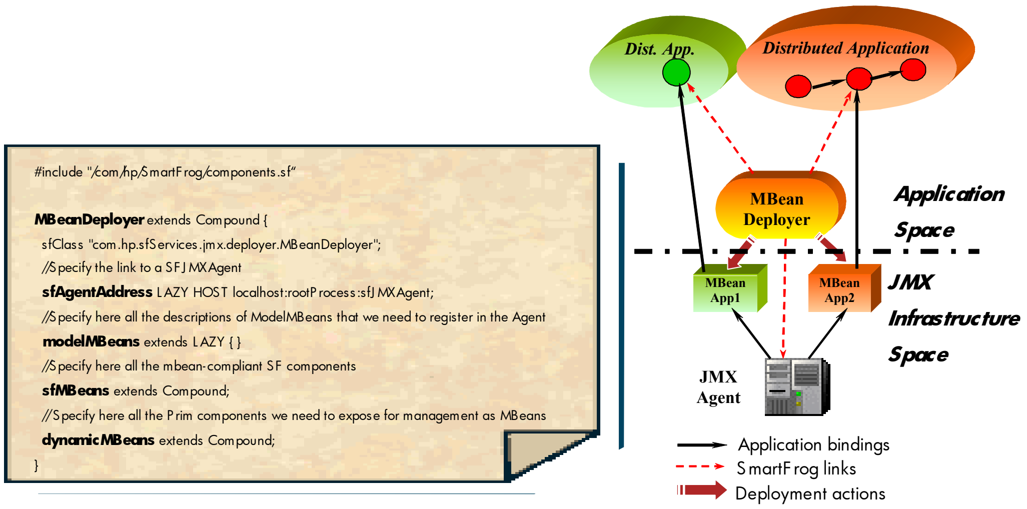
5 sfJMX Service Design

In the following sections we are going to describe briefly some of the most important components of the sfJMX service.

5.1 *MBeanDeployer*

The MBeanDeployer (Figure 4) is a SmartFrog component designed to allow the application designer to load all the required management information into a specific JMX Agent, providing enough transparency to the distributed application components. Thus application components may not need to include any line of code related to JMX.

Each MBeanDeployer is able to locate a JMX Agent and obtain from it the MBeanServer interface. It does not matter if the JMX Agent is remote to this MBeanDeployer, as long as the JMX Agent offers a ConnectorServer that gives access to the MBeanServer interface, for example the RMI connector provided with the sfJMX service



```
#include "/com/hp/SmartFrog/components.sf"

MBeanDeployer extends Compound {
  sfClass "com.hp.sfServices.jmx.deployer.MBeanDeployer";
  //Specify the link to a SFJMXAgent
  sfAgentAddress LAZY HOST localhost:rootProcess:sfJMXAgent;
  //Specify here all the descriptions of ModelMBeans that we need to register in the Agent
  modelMBeans extends LAZY { }
  //Specify here all the mbean-compliant SF components
  sfMBeans extends Compound;
  //Specify here all the Prim components we need to expose for management as MBeans
  dynamicMBeans extends Compound;
}
```

Figure 4. MBeanDeployer Description and Diagram.

The MBeanDeployer only takes part in the MBean deployment. Afterwards it does not participate in the management process but it remains as a simple proxy to access the JMX Agent to which it is attached. A sequence diagram for a common use case is shown in Figure 6.

As we can see in the Figure 6, it is the MBeanDeployer that is configured to locate the managed resource/s and instantiate, if necessary, the appropriate MBean/s in the JMX Agent. The MBeanDeployer configuration is provided by means of SmartFrog descriptions; hence, the application code does not need to know anything about the MBeanDeployer or JMXAgent.

As example, a partial description of an MBeanDeployer is shown in Figure 5. For a complete example refer to appendix.

```
//MBean Deployer
arithMBeanDeployer extends MBeanDeployer {
  //Agent where to register the MBeans
  sfAgentAddress LAZY HOST localhost:Agents:subAgent1;
  //SF components for whom to register MBeans in subAgent1
  sfMBeans extends Compound {
    leftGenerator LAZY ROOT:manager:leftGenerator;
    rightGenerator LAZY ROOT:manager:rightGenerator;
  }
  //Model MBeans to register in subAgent1
  modelMBeans extends LAZY {
      deploymentModelMBean extends DeploymentModelMBean {
        properties:name "ArithDeployment";
        managedResource LAZY ATTRIB deploymentAnalyzer;
      }
    }
  }
```

Figure 5. Example Description of an MBeanDeployer Component.

If preferred, the application itself can always take the initiative and register its MBeans at deployment or run time, using the MBeanDeployer as proxy to the JMX Agent.

The MBeanDeployer distinguish between 3 types of MBeans:

- sfMBeans: managed resources that implement the Standard MBean pattern, that is, provide a Java interface whose name has the suffix MBean.
- ModelMBeans: those resources that do not implement a Java interface as their management interface. A SF description has to be provided with all the necessary metadata.
- PrimDynamicMBeans: similar to the previous ModelMBeans but specialized for the SF component model.

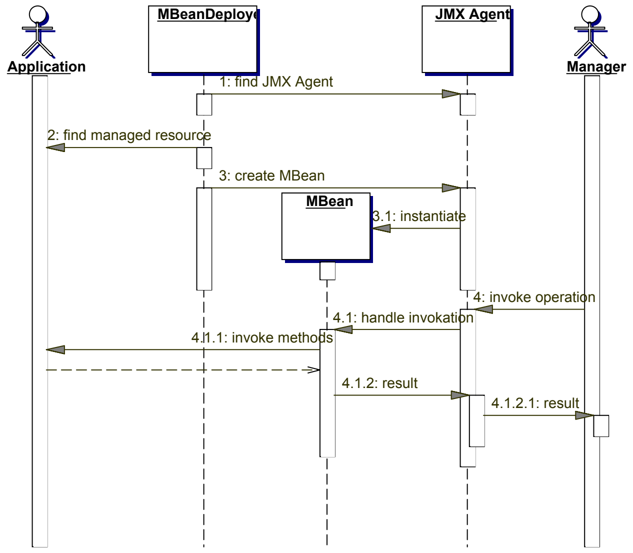
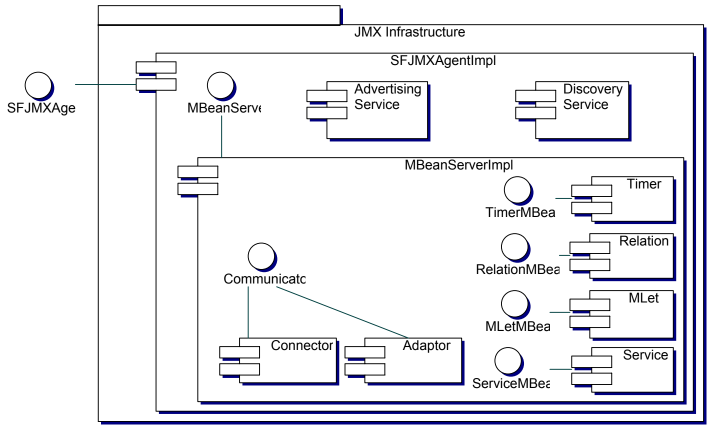


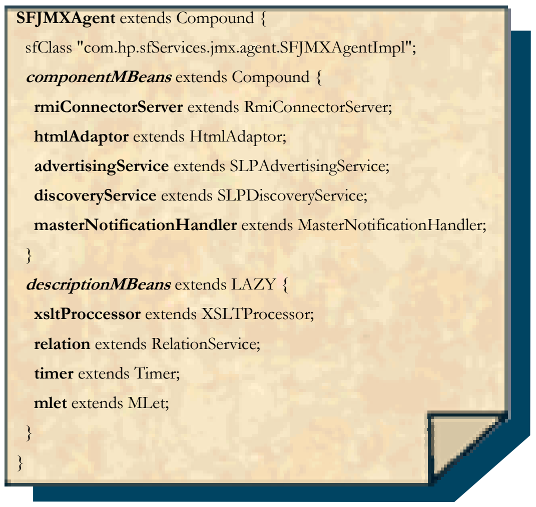
Figure 6. Sequence Diagram for a Common Use Case for MBean Deployment.

5.2 sfJMXAgent

The core of a JMX infrastructure is composed of a set of sfJMXAgents. A sfJMXAgent is an implementation of the Standard JMX Agent for the sfJMX package. Every sfJMXAgent can be seen as a composite subsystem, consisting of a set of elements that can be composed in different ways, according to the different uses. This is done using a SmartFrog description of the Agent. The structure of the Agent and its standard description template are shown in Figure 7.

sfJMXAgent (Figure 6) allows the user to specify in its SmartFrog description all the required MBeans that must be registered at deployment time. Typically this feature is used to plug and configure any kind of MBean that provides a service. This feature can also be used to deploy and register components of an application.





```
SFJMXAgent extends Compound {
 sfClass "com.hp.sfServices.jmx.agent.SFJMXAgentImpl";
 componentMBeans extends Compound {
   rmiConnectorServer extends RmiConnectorServer;
   htmlAdaptor extends HtmlAdaptor;
   advertisingService extends SLPAdvertisingService;
   discoveryService extends SLPDiscoveryService;
   masterNotificationHandler extends MasterNotificationHandler;
 }
 descriptionMBeans extends LAZY {
   xsltProccessor extends XSLTProcessor;
   relation extends RelationService;
   timer extends Timer;
   mlet extends MLet;
 }
}
```

Figure 7. Component Diagram and Description of the sfJMXAgent

The user can specify two types of MBeans:

- **MBean-compliant SmartFrog components**: they are SmartFrog components that implement a standard or dynamic MBean interface. They do not need to be instantiated because they had already been instantiated by the SmartFrog daemon. Since these components are SmartFrog components, they can configure themselves and be started as part of the normal SF component lifecycle.
- **Description MBeans:** they are just descriptions of MBeans that are not SF components, and thus they are not deployed by SF.

5.3 Advertising and Discovery Service

Advertising and Discovery services have been included within the MBeanServer so that they provide other MBeans with the capability to advertise and discover new services. This feature is used, for example, to discover and dynamically bind JMX Agents. These services can optionally be included in the MBeanServer and both make use of an SLP package provided by SmartFrog.

5.3.1 Advertising Service

There are two mechanisms to deploy this service: either using SmartFrog predefined components (SLP [4, 5]) properly configured to provide the service or using the Advertising Service MBean. Advertising is based on the IETF Service Location Protocol (SLP): Directory Agent and Service Agent (Advertiser).

The advertising service dynamically detects all available services and keeps a consistent set of service advertisements. It allows other MBeans to advertise services through its JMX interface. When the service is first started, it tries to find out which services of the distributed service level are active in the Agent at that moment and registers a proper advertisement in the Directory Agent.

5.3.2 Discovery Service

There are two mechanisms to provide a discovery service, the first one is by using SmartFrog predefined components and the second one is using the Discovery Service MBean. Discovery is based on SLP: User Agent (Locator).

Other MBeans are able to use the discovery service and for example perform a discovery of JMX Agents. The Monitoring Thread sends a Discovery Notification whenever an advertisement is registered or deleted in the Directory Agent

In the Agent, the Master Notification Handler MBean uses Discovery service to discover new JMX Agents and master them. As in the previous service, this one is also based on the SLP protocol.

The JMX interface provided by this service allows other components performs queries on the DAs available in the network to discover JMX related advertisements, it also allows other components to receive Discovery Notifications, which are emitted by a Monitoring Thread.

5.4 Cascader

The Cascader MBean enables the distributed JMX Agent topology. This MBean provides the means to create hierarchies of JMX Agents by cascading the information of lower level JMX Agents to upper level Agents (Figures 3 and 8). The upper level Agents can filter only that information which is of interest for a management application or other agent/s in an upper level. The hierarchy of agents is build following a Master-Subagent structure in which the Subagents don't know anything about the existence of JMX Master Agents.

When a Cascader MBean is instantiated, it will try to connect to a Remote Agent and perform a configurable query about the MBeans contained in that Agent. For every retrieved ObjectName, the Cascader MBean creates an MBean Proxy, which is a Dynamic MBean with the same ObjectName and that locally represents the remote MBean, hiding the remoteness of the real MBean. With this mechanism, the mastered Agent is not concerned about the existence of the Master Agent. This solution also allows introducing dynamic changes in the hierarchies and making it resilient to failures providing certain degree of high-availability.

To keep consistency between the remote MBeans and the local MBean Proxies, the Cascader MBean uses the Remote Notification System provided, so that when an MBean is registered or unregistered in either side, the Cascader tries to match that with the correspondent operation on opposite side.

This master-agent solution offers a similar approach to other JMX implementations, such as “AdventNet Agent Toolkit Java/ JMX Edition”[3]. As new features, sfJMX offers a way of describing and deploying complex JMX infrastructures and introduces the use of advertising and discovery services, which can be used, for example, for auto-configuration of the JMX Agent web.

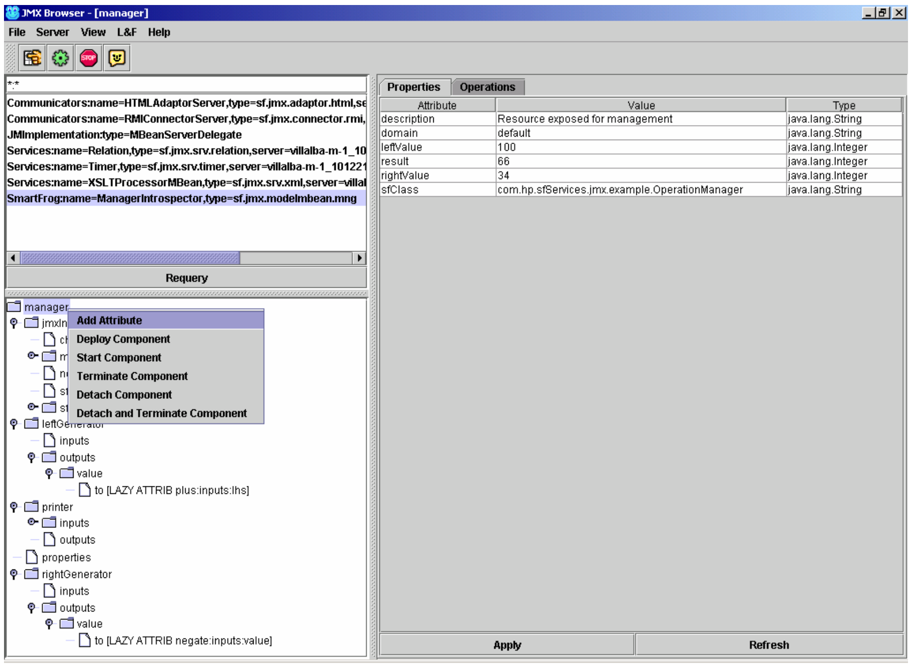
5.5 Additional Tools

Two additional tools are described in the next section.

5.5.1 JMX Browser

The MBeanBrowser (Figure 8) is a JMX-enable application that is able to show and administrate the MBeans registered in the Agent to which it is connected.

It enables a variety of actions to be invocated on the MBeans, such as to modify and monitor attributes, invoke operations or to create and unregistered MBeans.



```
*:*
Communicators:name=HTMLAdaptorServer,type=sf.jmx.adaptor.html,se
Communicators:name=RMIConnectorServer,type=sf.jmx.connector.rmi,
JMImplementation:type=MBeanServerDelegate
Services:name=Relation,type=sf.jmx.srv.relation,server=villalba-m-1_10
Services:name=Timer,type=sf.jmx.srv.timer,server=villalba-m-1_101221
Services:name=XSLTProcessorMBean,type=sf.jmx.srv.xml,server=villal
SmartFrog:name=ManagerIntrospector,type=sf.jmx.modelmbean.mng
```

Attribute	Value	Type
description	Resource exposed for management	java.lang.String
domain	default	java.lang.String
leftValue	100	java.lang.Integer
result	66	java.lang.Integer
rightValue	34	java.lang.Integer
sfClass	com.hp.sfServices.jmx.example.OperationManager	java.lang.String

```
manager
  jmxIn
    ch
    m
    n
    st
    st
  leftGenerator
    inputs
    outputs
      value
        to [LAZY ATTRIB plus:inputs:lhs]
  printer
    inputs
    outputs
  properties
  rightGenerator
    inputs
    outputs
      value
        to [LAZY ATTRIB negate:inputs:value]
```

Figure 8. MBeanBrowser

In cooperation with the Deployment Analyzer, the MBeanBrowser detects those MBeans that are Deployment Analyzers and perform the necessary operations on them to display the deployment tree of SF applications.

5.5.2 Deployment Analyzer

This MBean provides a set of methods that help other MBean browse down the SmartFrog deployment tree of an SF application. It also provides direct access to the APIs of each SF component deployed. The deployment Analyzer will only expose for management the data specified as manageable metadata.

6 Conclusions

Concluding, the sfJMX service provides a full solution from the managed resource side to the administrator side, ready to be included within application descriptions without writing a single line of code, simply by adding some components and specifying the metadata that describes the information that the designer would like to have available for management. Nevertheless, the designer can make use of the sfJMX and the JMX-RI API conforming to his needs. The sfJMX service also provides the necessary components required to manage large-scale applications, such as Cascader and advertising and discovery service. With these extensions it is possible to create distributed hierarchies of Agents and use discovery as a dynamic binding between them. The new service for service advertising, lookup and discovery is also available to external applications.

Future work in the framework should be focus on designing a security model for accessing the MBeanServer and a security model for JMX in general.

7 Acknowledgements

Thanks to Antonio Lain, Guillaume Mecheneau, Paul Murray and Peter Toft, integral part in the team developing the SmartFrog framework and thanks to Jorge López de Vergara for his valuable comments.

8 References

1. Sun Microsystems. *Java Management Extensions Instrumentation and Agent Specification, v1.0*. July 2000, Sun Microsystems. http://java.sun.com/jmx.
2. Juha Lindfors, Marc Fleury. *JMX. Managing J2EE with Java Management Extensions*. 2002 Sams Publishing.
3. AdventNet, Inc., http://www.adventnet.com/.
4. James Kempf, Pete St. Pierre. Service Location Protocol for Enterprise Networks. 1999 Wiley Computer Publishing.
5. SLP information: http://www.openslp.org/.
6. JSR 160, JMX 1.5, http://jcp.org/jsr/detail/160.jsp.
7. SmartFrog Reference Manual. 2001 HP Laboratories. Bristol, UK.
8. Patrick Goldsack. “SmartFrog, a framework for configuration”. E-science workshop, Nov. 2001. Edinburgh University. http://www.dcs.ed.ac.uk/home/dcspaul/wshop/SmartFrog.pdf

9. Juan I. Asensio, Víctor A. Villagrá, Jorge E. López·de·Vergara, Roney Pignaton, Julio J. Berrocal. “Experiences in the management of an EJB-based e-commerce application”. HP Openview Univesity Association Eighth Plenary Workshop (HP-OVUA'2001). Berlin, Germany. June 2001.

Appendix

This appendix contains a couple of descriptions for a JMX infrastructure and an example application using SmartFrog notation. A representation of the example is shown in Figure 9.

The JMX infrastructure (Figure 10) is a simple master-subagent structure. The application MBeans are registered in the SubAgents. The Master Agent acts as a single “gateway” to manage the entire application.

The example application (Figure 11) implements a simple arithmetic service. This service is formed by several components interconnected. There are two number generators (1 and 2), two operations (+,-) and a printer component to show the final result of the operation. Additional components are created to instrument the application. There is a management component that contains references to the number attributes of the generators, the result attribute of the printer and a special operation to feed the arithmetic net. This management component is registered in the JMX SubAgent 2 and accessed through the Master Agent. In the description is shown how the JMX metadata for attributes and operations is described using SF notation. The two generators and the printer components are also registered as MBeans. A SmarFrog Deployment Analyzer is also included in the description to get access to SF environment API’s.

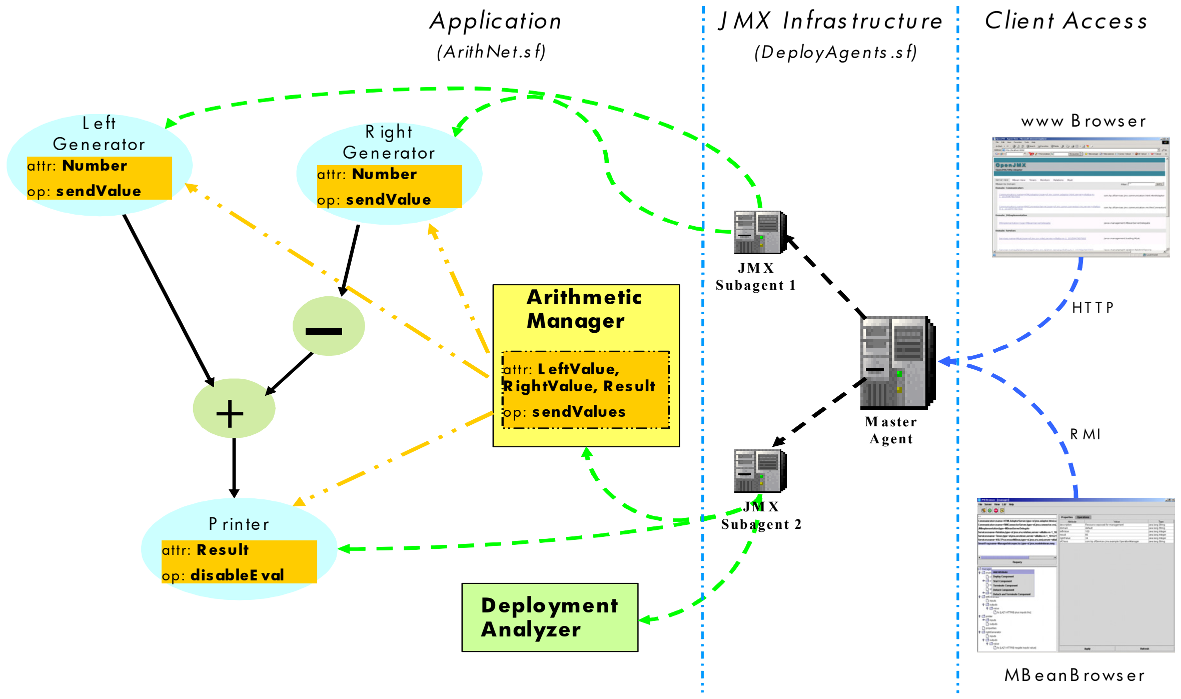


Figure 9. Example application: Arithmetic Net.

Description for JMX infrastructure “DeployAgents.sf”:

```
#include "host.sf"
#include "/com/hp/SmartFrog/components.sf"
#include "/com/hp/sfServices/jmx/components.sf"

// SubAgent1
SubAgent1 extends SFJMXAgent {
  sfProcessHost ATTRIB host1;
  componentMBeans:advertisingService "";
  componentMBeans:htmlAdaptor:port 8081;
  componentMBeans:rmiConnectorServer:properties:name "Subagent1";
}
// SubAgent2
SubAgent2 extends SFJMXAgent {
  sfProcessHost ATTRIB host2;
  componentMBeans:advertisingService "";
  componentMBeans:htmlAdaptor:port 8082;
  componentMBeans:rmiConnectorServer:properties:name "Subagent2";
}

// MasterAgent
MasterAgent extends SFJMXAgent {
  componentMBeans:advertisingService "";
  componentMBeans:cascader1 extends Cascader {
    properties:name "Cascader1";
    host ATTRIB host1;
    resource "Subagent1";
  }
  componentMBeans:cascader2 extends Cascader {
    properties:name "Cascader2";
    host ATTRIB host2;
    resource "Subagent2";
  }
}

//Components Deployment
sfConfig extends Compound {
  subAgent1 extends SubAgent1;
  subAgent2 extends SubAgent2;
  masterAgent extends MasterAgent;
}
```

Figure 10. Description "DeployAgents.sf".

Description for application deployment “ArithNet.sf”:

```
#include "/com/hp/SmartFrog/components.sf"
#include "/com/hp/sfServices/jmx/components.sf"
#include "/com/hp/sfServices/jmx/example/components.sf"
#include "/com/hp/sfExamples/arithNet/netComponents.sf"

// Attributes Metadata
LeftValue extends AttributeInfo {
  name "leftValue";
  description "The current value in the left generator";
}

RighValue extends AttributeInfo {
  name "rightValue";
  description "The current value in the right generator";
}

Result extends AttributeInfo {
  name "result";
  writable false;
  description "The last result obtained in the arithmetic net";
}

// Operations Metadata
SendLeftValue extends OperationInfo {
  name "sendLeftValue";
  impact "ACTION";
  description "Force the left generator send its current number to its output";
}

SendRightValue extends OperationInfo {
  name "sendRightValue";
  impact "ACTION";
  description "Force the right generator send its current number to its output";
}

SendBothValues extends OperationInfo {
  name "sendBothValues";
  impact "ACTION";
  description "Force both generators send their current number to their outputs";
}

//Extended Metadata for Deployment Analyzer.
ArithManageableMetaData extends ManageableMetadata {
  leftValue extends LeftValue;
  rightValue extends RighValue;
  result extends Result;
  number extends AttributeInfo {
    name "number";
    description "The current value in this generator";
  }
  sendLeftValue extends SendLeftValue;
  sendRightValue extends SendRightValue;
  sendBothValues extends SendBothValues;
}

//Components Deployment
sfConfig extends Compound {
  //Special SF Component to expose SF API
  deploymentAnalyzer extends DeploymentAnalyzer {
    sfManageable extends LAZY ArithManageableMetaData;
  }
  //MBean Deployer registering Standard MBeans
  arithMBeanDeployer1 extends MBeanDeployer {
    sfAgentAddress LAZY HOST localhost:Agents:subAgent1;
    sfMBeans extends Compound {
      leftGenerator LAZY ROOT:manager:leftGenerator;
      rightGenerator LAZY ROOT:manager:rightGenerator;
    }
  }
  //MBean Deployer registering model MBeans and SF component as MBeans
  arithMBeanDeployer2 extends MBeanDeployer {
    sfAgentAddress LAZY HOST localhost:Agents:subAgent2;
    modelMBeans extends LAZY {
      deploymentModelMBean extends DeploymentModelMBean {
        properties:name "ArithDeployment";
        managedResource LAZY ATTRIB deploymentAnalyzer;
      }
    }
    sfMBeans extends Compound {
      arithManager LAZY ATTRIB manager;
      printer LAZY ROOT:manager:printer;
    }
  }
  //Real application Manager
  manager extends ArithmeticManager {
    // Link generator outputs to operator inputs
    leftGenerator:outputs:value:to LAZY ATTRIB plus:inputs:lhs;
    rightGenerator:outputs:value:to LAZY ATTRIB negate:inputs:value;
  }
  //Negate operator
  negate extends Negate {
    outputs:value:to LAZY ATTRIB plus:inputs:rhs;
  }
  //Addition operation
  plus extends Plus {
    outputs:value:to LAZY ATTRIB manager:printer:inputs:printVal;
  }
}
```

Figure 11. Description "ArithNet.sf".