Livrable Selfware SP1

Lot 4, December 16th, 2008

Selfware Common Services for Self-Management, version 2

Authors
Fabienne Boyer (INRIA/Sardes)
Bruno Dillenseger (France Telecom)
Daniel Hagimont (IRIT/ENSEEIHT)
Mayleen Lacouture (EMN)
Thomas Ledoux (LINA/EMN)
Benoît Pelletier (Bull)

Version 1.3

Contents

1 Introduction

Autonomic computing, which aims at the construction of self-managing and self-adapting computer systems, has emerged as an important goal for many actors in the domain of large scale distributed environments and applications. This approach more precisely aims at providing systems and applications with self-management capabilities, including self-configuration (automatic configuration according to a specified policy), self-optimization (continuous performance monitoring), self-healing (detecting defects and failures, and taking corrective actions), and self-protection (taking preventive measures and defending against malicious attacks).

Following this approach, the Selfware project aims at providing an infrastructure for developing autonomic management software. An important aspect of this infrastructure is the adoption of an *architecture-based* control approach as described in the SP1-L1 document, meaning that the control loops that regulate the system have the ability to introspect the current software architecture of the managed system, as well as they have the ability to modify (i.e. reconfigure) this architecture.

The introspection and reconfiguration capabilities that can be invoked within control loops mainly rely on the *common services* provided by the Selfware infrastructure. These services act on a managed system. Examples of these services are a deployment service, for deploying managed legacy software in a distributed environment, a monitoring service for observing a given managed element, or a scripting language for dynamically exploring the software architecture of a managed system.

The objective of this document is to give an update of the previous deliverable dedicated to the common services provided by the Selfware platform (SP1, Lot 2). A short summary of the previous services is first given as a reminder, and then, the evolution that has been made as a follow-up to improve the Selfware platform.

The software parts associated to the present document are mainly available on the site http://wiki.jasmine.objectweb.org/ (other parts' availability is given inline).

2 Feedback from common services version 1

2.1 Summary of common services version 1

The Selfware common services version 1 are described in deliverable document SP1L2. They provide a set of basic capabilities used by autonomic managers to perform elementary configuration management functions, such as installing and deploying components on nodes, or communicating with sensors and actuators. Since common services are mainly built themselves with components, they may also be considered as managed elements in the managed system. The common services that are provided by version 1 of the Selfware infrastructure are:

- The wrapping service, that allows generating the wrappers used to control legacy software.
- The navigation service designed to express queries on the managed system's architecture.
- The reconfiguration service, used to define consistent reconfiguration on the managed system's architecture.
- The resource allocation service, that allows allocating resources (e.g. nodes) for the managed system as well as for the management system.
- The deployment service, that aims at deploying both the managed system and the management system on remote nodes.
- The monitoring service used to gather information on the managed system and to aggregate these information to provide high-level events attached to more semantic.
- The system representation service, used to add reliability to the configuration actions performed by the Autonomic Managers by replicating the critical components.
- The decision service used to implement the reactive part of autonomic managers.

Next section is focusing on some key services that appeared to deserve a peculiar attention and probably some more work.

2.2 Feedback

2.2.1 Monitoring support

Selfware actually handled two monitoring frameworks. Each of them focused on some issues, and probably missed some points.

Composite probes (CPs) propose a pure-component-based framework that focuses on the architectural aspects of probes. CPs define both the inner architecture of probes, identifying and encapsulating a number of probe sub-functions (measuring, aggregating, filtering, storing), and the external architecture in terms of probe composition. Then, CPs address two key motivations:

- be fully inline with Selfware's component-based architectural approach to autonomic computing. This results in a better integration to the Selfware platform, since the monitoring service itself can be considered as a Managed Element, and then benefit from the common services, such as deployment or self-repair;
- provide an elegant and uniform way of building arbitrary combinations of probes through composite probes, to support any kind of high-level monitoring feature (history consulting, aggregations and filtering of any sort, etc.). This is achieved thanks to a particular ability to compose the same running probes in several assemblies providing different monitoring features (either through bindings or containment relations).

On the negative side, CPs don't provide a high-level API for developers, in order to easily interact with the monitoring service (interrogate, listen, put some triggers on specific events, etc.). Moreover, it also misses consideration of the event transport concerns. Events (measures) transport is based on

Fractal bindings, typically using FractalRMI for distributed communication. To go further, the CPs framework should be enhanced with some special bindings that would support typical event bus/switch features, with some publish/subscribe mechanism. The seamless integration of a message-oriented middleware would provide better performance, efficiency, and practical manageability with regard to event transport.

WildCAT is a monitoring middleware that provides an extensible Java framework to ease the creation of context-aware applications. WildCAT provides a simple yet powerful dynamic model to represent an application’s execution context. The context information can be accessed by application programmers through two complimentary interfaces: synchronous requests (pull mode) and asynchronous notifications (push mode). On the negative side, WildCat does not adopt a Fractal architecture, which both limits the readability of its architecture, and its manageability by the Selfware platform's common services. Moreover, just like CPs do, WildCat does not address distribution and event transport issues in a satisfactory manner.

JMX is a so-called Java eXtension for Management. As of today, JMX is a well supported standard in Java technologies, and most of Java Application Servers, such as JOnAS, provide so-called MBeans servers to provide monitoring and management facilities. However, using the JMX API is a bit cumbersome and leads to long, poorly readable and hard to maintain pieces of Java code. In a very similar way that FPath (coming along with FScript, see deliverable SP1L2) does for browsing and searching Fractal components, there is a need for a declarative, expressive way of designating and looking for MBean objects according to their name or attribute values.

2.2.2 High-level administration tools based on DSLs

The Domain-Specific Languages approach has been successfully applied to automate the burden of writing wrappers to legacy software. The provided DSL gives a declarative support for generating Fractal wrappers that encapsulate necessary primitives for configuring, instantiating, starting and stopping legacy software. But it appears that the DSL approach could be useful to other Selfware features. For instance, current Fractal ADL could be enhanced so that an architecture description can be defined in an intension rather than extension way, describing some abstract architectural patterns rather than one-for-one component description. Another example is about a declarative support for reconfiguration actions that may be taken by autonomic managers.

2.2.3 Decision support

A framework has been proposed to implement high-level rules based on the Event-Condition-Action model, inherited from the active databases domain. This highly adaptable framework relies on an advanced and elegant architectural integration of ECA rules in the form of components, taking advantage on Fractal's enhanced features (e.g. with a wide use of component hierarchy and sharing support). It exhibits and allows customizing the full set of policy choices with regard to triggering, evaluating and executing the rules. However, the practical use of this ECA framework tackles a lack for some higher-level API and tooling in order to ease the definition of rules and execution policies.

2.3 Conclusion

The first generation of Selfware services is typically motivated by software architecture concerns, with a close relation to the Fractal component model: the system repository, ECA rules, Fractal wrappers, composite probes, Fscript and FPath... Then, it appears that these fundamental services layer provide a basic, kind of low-level component infrastructure, but still require higher-level APIs and tools to be of effective usability by systems and applications administrators. The requirement is to simplify the definition and deployment/integration of specific rules, policies, monitoring facilities in a particular environment, for particular applications. WildCAT was the first illustration of this need.

The next section focuses on the several new facilities addressing general monitoring needs. Then the section after gives a twofold description of enhanced design and management tools, with a generic software engineering toolbox on the one hand, and a J2EE-dedicated toolbox on the other hand.

3 Enhanced monitoring facilities

3.1 Monitoring support with WildCAT v2

3.1.1 Rationale

To overcome some limitations of WildCAT v1, we decided to rewrite the kernel of the framework to build a new version around an open source CEP (Complex Event Processing) engine. WildCAT v2 is based on a backend called Esper (http://esper.codehaus.org). This new version permits the monitoring of large scale applications by allowing developers to easily organize and access sensors through a hierarchical organization backed with a powerful SQL-like language to inspect sensors data and to trigger actions upon particular conditions.

3.1.2 Modelling Data

Hierarchical Data Source Organization

WildCAT let developers organize data sources inside a Context. A WildCAT context is an oriented tree structure with two types of nodes:

- Attributes nodes that hold some value. Attributes nodes are always leaf nodes, and every attributes has a unique parent node. There exists 3 types of attributes:
    - Basic attributes holds static values. Their values do not evolve unless programmatically modified.
    - Active attributes (sensors)
    - Synthetic attributes are the results of expressions on other attributes
- Resources nodes that can have zero or more children assuming every child name is unique. A resource may have more than one parent. There are two types of resources:
    - Basic resources.
    - Symbolic links are special resource that alters path resolution by pointing to another resource.

The example hierarchy in figure 1 presents three resources (blue circles) and five attributes (red squares). Every WildCAT Context has a unique entry point: the ROOT resource. Moreover a Context must never contain a cycle.

When manipulating a WildCAT context, developers reference resources and attributes by their path from the ROOT resource. WildCAT paths are a convenient way for users to denote resources and attributes.

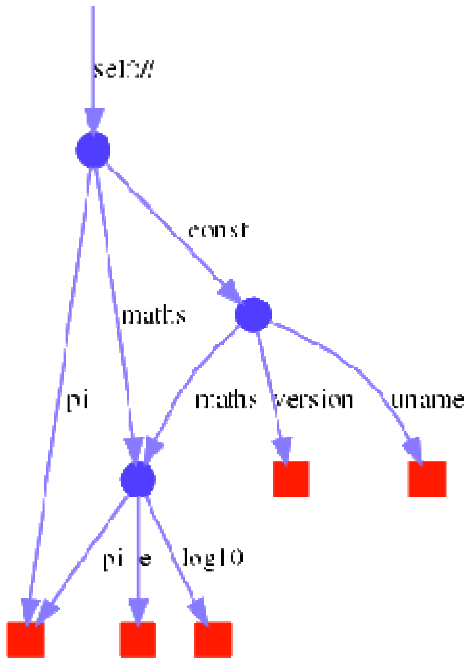


figure 1. WildCAT Hierarchical Organization

Exception made from the ROOT node, a node may be resolved through multiple paths: in our example, both paths "self://consts/maths" and "self://maths" denotes the same resource and the bottom left attribute may be resolved by "self://#pi", "self://maths#pi" or "self://consts/maths#pi".

Paths ending with "#name" denote attributes. The context name "self" is a reserved name that is an equivalent of "localhost" in networking".

The Context interface

The Context interface is the one and only entry point for WildCAT developers to organize, inspect and register notification handlers on context data provided by sensors. WildCAT provides two different ways of instantiating the Context interface: (i) Instantiating the BasicContext class; (ii) Creating a Context through a ContextFactory. To go further, see http://wildcat.objectweb.org.

A default ContextFactory is accessible in the ContextFactory class, through the static method **`getDefaultFactory()`**. It is the easier way to create a new context.

```
Context ctx = ContextFactory.getDefaultFactory.createContext();
```

3.1.3 Inspecting Data

WildCAT event model

WildCAT permits developers to inspect the content of a Context by registering queries on the event generated by the hierarchy. In WildCAT every events implements the WEvent interface. It allows determining for every event, the node in the hierarchy which emitted the event. There are two kinds of event emitted by a WildCAT context. Firstly, events emitted by Resources: they instantiate the WHierarchyEvent class and informs about the operation performed on the structure of the hierarchy. For example, such an event may indicate the addition of a attribute named "fs" to the resource "self://proc". Secondly, events emitted by Attributes instantiate the WAttributeEvent class and indicate the modification of that attribute and holds its new value.

PULL mode (synchronous)

In PULL mode, developers programmatically get and set attributes in a synchronous way. The Context interface provides methods to manipulate the WildCAT hierarchy, ie. creating resources and attributes, getting and setting attribute's values. For example:

```
ctx.createAttribute("self://constants#hello", "Hello");
```

creates an primitive attribute with initial value "Hello". In the path "self://constants#hello", "self://" denotes the current context, "constants" is a resource mounted at top level, and the "hello" identifier following the '#' sign, is the name of the attribute attached to the resource "constants". Despite the resource "constants" did not exists beforehand, the "createAttribute" method will create every missing resource along the path as long as it complies with the WildCAT hierarchy constraints (attributes and resources attached to a given resource must be unique). Once created,

```
System.out.println("self://constants#hello = " + ctx.getValue("self://constants#hello"));
```

would output: "self://constants#hello = Hello". Now, let's change that attribute value:

```
System.out.println("self://constants#hello = " + ctx.setValue("self://constants#hello", "Hello
World !!!"));

System.out.println("self://constants#hello = " + ctx.getValue("self://constants#hello"));
```

the first statement has the same output: "self://constants#hello = hello". Indeed, the "setValue" method of the Context interface returns the previous value of the attribute. After executing the "setValue" method, the second statement outputs "self://constants#hello = Hello World !!!".

Symbolic links are special resource that holds a reference to another resource. Symbolic links transparently alters path resolution by pointing to another resource. The Context interface allows for easy creation of symbolik links, the following code:

```
ctx.createSymbolicLink("self://demo/soft/link/toConstant", "self://constants");
```

creates a symbolic link at position "self://demo/soft/link/toConstant" pointing to "self://constants". From now every path resolution entering resource "self://demo/soft/link/toConstant" will be redirected to "self://constants".

PUSH mode (asynchronous)

In PUSH mode, developers register listeners on queries expressed over the event generated in the Context. The Context interface provides operations to create queries and to register listeners on queries. To perform query processing, WildCAT relies on the Esper Complex Event Processing Engine (http://esper.codehaus.org)[1]. Queries are described in the Event Query Language (EQL), a SQL-like language with `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` and `ORDER BY` clauses. Streams replace tables as the source of data with events replacing rows as the basic unit of data.

The following code:

```
Query query = ctx.createQuery("select * from WEvent");
```

returns a query reference for **select * from WEvent**. This query will catch every event traversing WildCAT. WildCAT automatically starts queries upon creation. In our example, we must now attach a listener to the newly created query. The Context Interface provides the registerListeners method that takes a Query instance and an Array of UpdateListeners (Varargs style) to attach. The most convenient way to use that method is to create an anonymous implementation of the UpdateListener interface:

```
ctx.registerListeners(query, new UpdateListener () {
    void update (EventBean[] nevents, EventBean[] oevents) {
        System.out.println("Received events");
    }
});
```

[1] More precisely, WildCAT 2.0 is based on Esper 1.12

Esper EQL is powerful and we can write complex and interesting queries. For example, the following query:

```
select avg(value('load')?) from WAttributeEvent(source='self://proc/cpu#info').win:length(5s)
```

is triggered on every WAttributeEvent emitted by attribute "self://proc/cpu#info" and returns the average value of the property "load" over every such events in the last five seconds.

Another example, the following query:

```
select * from pattern[every A=WAttributeEvent(source = 'self://date#time', value('second')? <
30)]
```

is triggered on every A event (i.e. select only attributes with the 'second' property value < 30).

WildCAT provides means to create attributes that holds the result of query. These special attribute called "query" attributes, are associated with a query. Query attributes can be used as normal attributes, i.e. their content can be accessed in PULL mode, it returns the last result of the query; and one can reference them in PUSH mode as part of a query. Result is dynamically updated. The following code snippet creates such an attribute:

```
ctx.createQueryAttribute("self://the/new/query#attribute", "select min(value(second)?) from
WAttributeEvent(source='self://date#time').win:length(5)");
```

The new attribute "self://the/new/query#attribute" will hold the minimum value of the property "second" of the attribute "self://date#time" over the last 5 seconds.

3.1.4 WildCAT Sensors

The sensors library (as of july 2008)

WildCAT team main objective is not to develop sensors, but to provide a generic framework to organize and inspect monitoring sources. Nevertheless, we distribute along with the WildCAT framework a sensors library that we extend as we develop new sensors for our own experiment.

- Java related sensors
    - JavaRuntimeSensor provides information relative to the current JVM (processor, memory, etc)
    - SystemPropertiesSensor provides a wrapper for Java System Properties
    - DateTimeSensor provides system current time
    - MBeanCMDSensor allows for gathering of any JMX related information (defined in synergy with the JASMINe project)
- System related sensors (Linux-only)
    - KernelVersionSensor provides system kernel version
    - CPUSensor provides information (model, version, etc) of the CPU
    - CPULoadSensor provides instant CPU load

Extending the sensors library

WildCAT's sensors must all implements the Attribute interface and should be attached to a WildCAT's hierarchy using the "attachAttribute" from the Context interface:

```
Context ctx = ContextFactory.getDefaultFactory.createContext();

Attribute mySensor = new MySensor();

context.attachAttribute ("self://path/to/my#sensor", mySensor);
```

To develop the class MySensor, we have to subclass the POJOAttribute class, which is the most common type of attribute and overload the getValue /setValue methods. Implementing this sensor is a matter of very few lines of code. Nevertheless, those sensors trigger new values only when one inspect its content. That sensor is said to be *passive*. To overcome this limitation, one may create a periodical attribute poller. Indeed, WildCAT provides means to simulate an *active* sensor based on a passive one by periodically crafting an event with the current value a given attribute. The following code snippet will create an new event containing current value of attribute "self://path/to/an#attribute" every 2 seconds:

```
ctx.createPeriodicAttributePoller("self://path/to/an#attribute", 2, TimeUnit.Second);
```

3.1.5 Distributed Contexts

In addition to the hierarchical organization of data source and PUSH/PULL operations, WildCAT allows for mixing PUSH and PULL and to connect distributed contexts.

The following figure (figure 2) introduces a more complex usage of WildCAT's contexts. Again, basic resources are represented by blue circles and red squares marks basic attributes. In addition, red triangles indicate special "query" attributes, and yellow polygons represent symbolic links. Finally groups of nodes bordered in blue represent another WildCAT instance (possibly on a remote host).

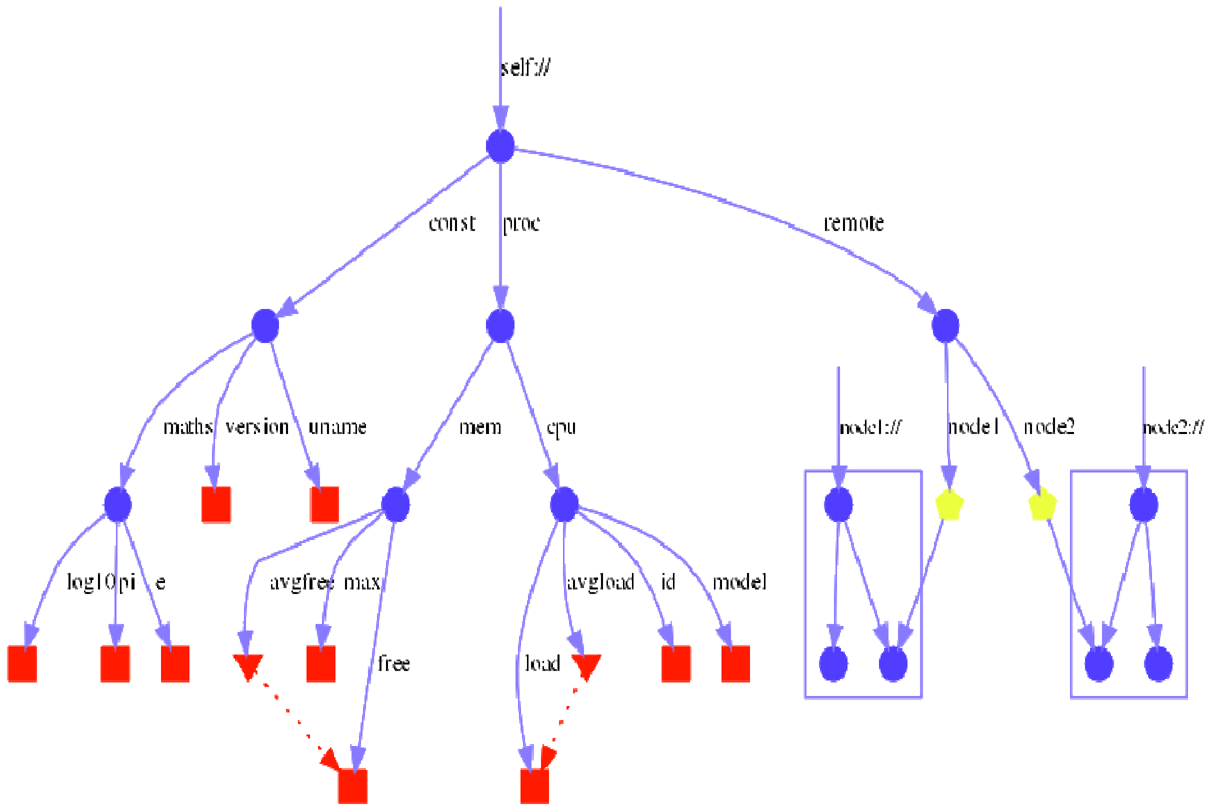


figure 2. Distributed contexts

WildCAT permits to connect remotes context using JMS and RMI technology. While one can directly inspect a remote context using the PULL API with an explicit remote path, WildCAT also permits the creation of symbolic links across contexts. PULL request over symbolic links are carried through RMI, while PUSH notification are done over JMS.

Remote contexts are accessed through *dispatchers*. Two alternative dispatchers are provided: one based on both RMI and JMS (used by default), and another fully implemented with JMS. If these dispatchers have the same implementation of push mode, they differ over the pull mode. Indeed, the RMI-JMS dispatcher uses RMI through CMI to perform synchronous operations, while the full-JMS dispatcher defines a layer to add synchronism on top of JMS.

3.1.6 Conclusion

Unlike WildCAT 1.0, WildCAT 2.0 provides (i) a strong support for distribution; (ii) a new event-based-model and a query language, based on an open source CEP (Complex Event Processing) engine. This new version was finished in august 2008 and was integrated by Bull partner in JASMINe platform.

We can point out that WildCAT 2.0 is embedded in the galaxy platform (http://galaxy.gforge.inria.fr), an open SOA platform enabling agility using dynamic architectures.

3.2 Easy JMX MBeans browsing with FPATH-JMX

3.2.1 Rationale

Despite the advantages offered by the model, Fractal is nowadays only used in some existing middleware platforms, while technologies like JMX (Java Management Extensions) are more commonly used in real-life solutions. Such is the case of application servers like JBoss and JOnAS.

The objective of this extension is to offer the language support provided by FPath (see previous Deliverables) in systems already using JMX as management platform. In that way, administrators will be able to navigate through MBean Servers to monitor the application's resources.

3.2.2 JMX Overview

JMX (Java Management Extensions) is a Java API to manage and monitor resources such as applications, devices, services or even the JVM (Java Virtual Machine). Each resource is instrumented by one or more managed Beans or MBeans, which are themselves Java objects, similar to Java Beans components. MBeans are registered in an MBean server, which is a management agent that runs on any Java enable device.

MBeans implement access to resources through a management interface consisting of attributes that can be written or read; operations, that can be invoked; and notifications, which are emitted by the MBean when some modification occurs.

3.2.3 FPath-style JMX Model

The JMX model implemented in the solution describes JMX architectures in the terms used by FPath: axes and nodes. The diagram in figure 3 shows the JMX elements and their corresponding relations as instances of FPath's nodes and axes.

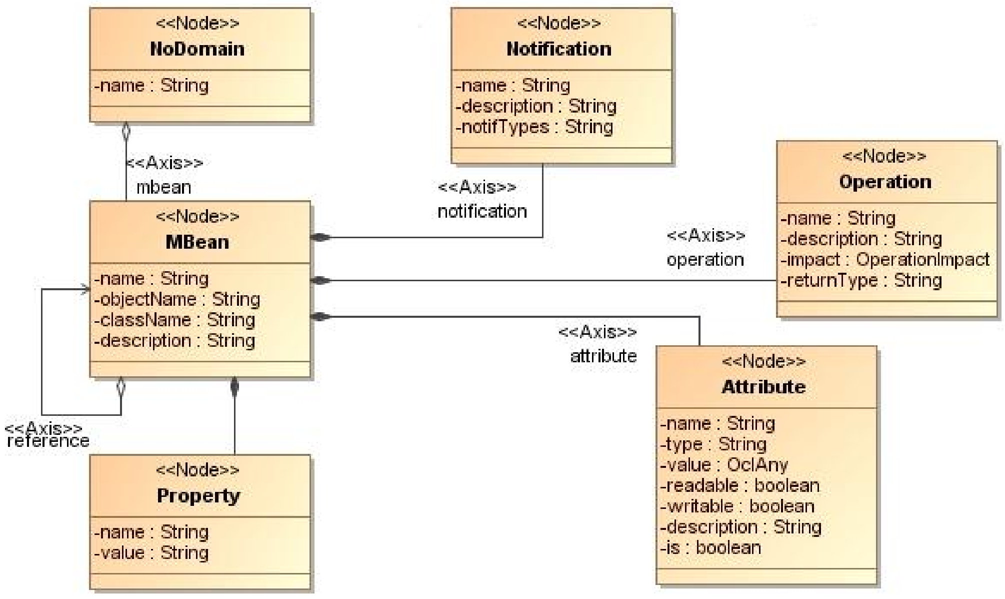


figure 3. JMX Meta model diagram

The nodes of the JMX model are:

- mbean node, represents the core element of the JMX technology.
- attribute node, represents the configuration attributes exposed by Mbeans. They have a type, a name and a value. As MBeans, they have a description and some additional properties to indicate whether they are readable, writable or a boolean expression.
- property node, corresponds to the MBeans' key properties that identify and classify the MBeans.
- domain node, represents the JMX domains in which MBeans are registered within the MBean Server.
- operation node, represents the operations exposed by MBeans, which are also part of the management interface. They have a name and a return type, also a description and some information about their impact when invoked, whether it is an action, action-info, info or unknown operation.
- notification node, represents the notifications which are emitted by MBeans when certain events occur. In the model, the notification node contains the name, description and type of a notification object.

The axis connecting these nodes are: mbean axis, which allows to select mbean nodes from domain nodes; attribute, property, notification and operation, which allow to select, respectively: attribute, property, notification and operations nodes from mbean nodes.

Finally, although they do not make part of the model itself, the JMX extension provides procedures like new-connection() and domains(), which allow to connect to the MBean Server and to obtain the domains registered in that server.

3.2.4 Examples

The test cases presented here where applied over the JOnAS application server.

Connection to the MBean Server

To access the MBean is necessary to create a connection to the MBean Server in which MBeans are registered. The procedure new-connection() initializes the connection to the URL specified as parameter, all queries in the session will be performed over that connection.

```
new-connection("//service:jmx:rmi://host/jndi/rmi://host:1099/jrmpconnector_jonas");
```

Next, it is necessary to execute the domains() function to obtain the domain beans corresponding to all domains registered in the MBean server, and save then in a new variable d.

```
d = domains();
```

If we want to list the domains registered in the server we execute the query

```
$d/domain::*
    #<domain: jonas>
    #<domain: Joram>
    #<domain: JMImplementation>
    #<domain: connectors>
    #<domain: AgentServer>
    #<domain: joramClient>
```

Querying queues by their depth

For this example, we want to know which MBeans instrument message queues with a depth greater than 5. For that, we execute a query in three steps. The first one corresponds to the set of all domains registered in the server. The second one, domain::joramClient selects joramClient domain. And finally, mbean::* takes all MBeans from the domain. Two predicates are applied to filter the result, [./property::type[value(.)=='queue']] selects only MBeans with a property type=queue, and [./attribute::PendingMessages] selects MBeans with an attribute PendingMessages greater than 5.

```
$d/domain::joramClient/mbean::*[./property::type[value(.)=='queue']]
                                [./attribute::PendingMessages[value(.) > 5]]
```

This query can be reduced to:

```
$d/domain::joramClient/mbean::*[mbtype(.)=='queue']
                                [./@PendingMessages[value(.) > 5]]
```

using the @ shortcut for the attribute axis, and the function mbtype(.), which replaces: value(./property::type).

Querying the Virtual Machine

With FPath is also possible to query the MBeans that instrument the JVM (Java Virtual Machine). To connect to the JVM, the URL parameter of function new-connection changes to local-vm.

```
new-connection("local-vm");
```

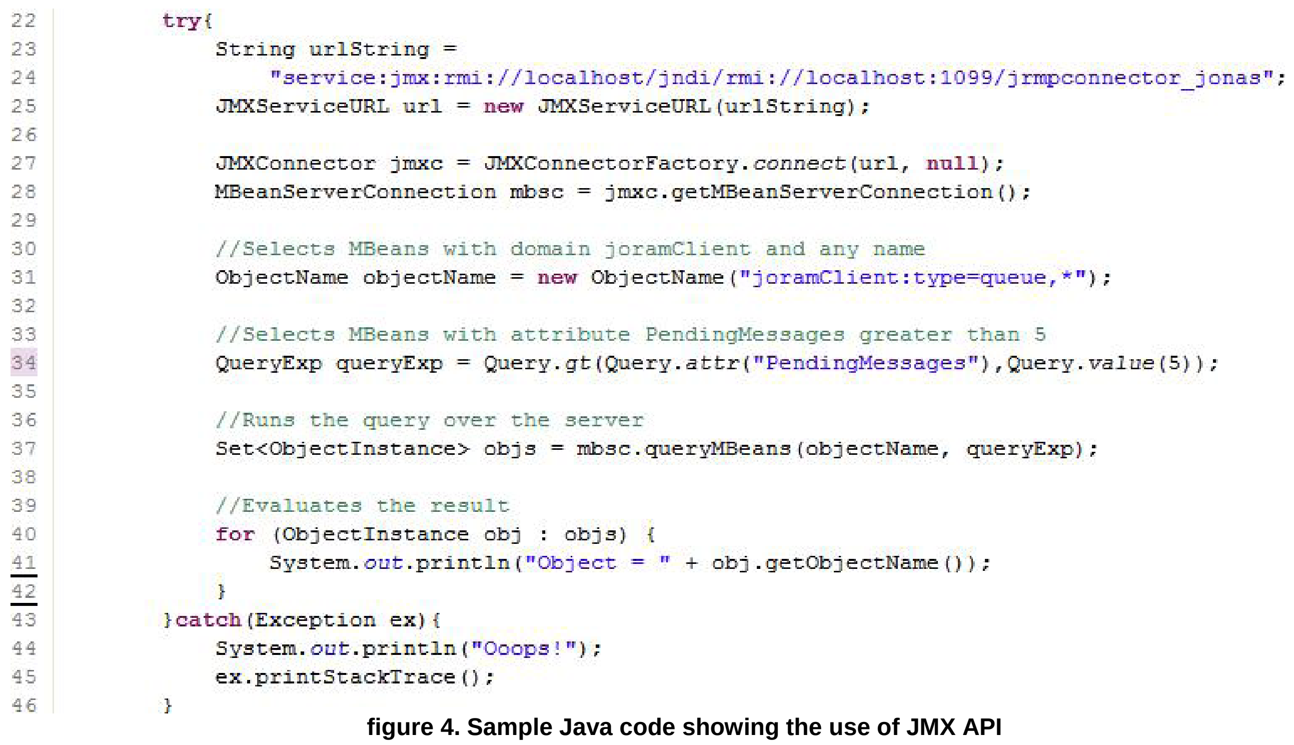
Assume we want to know when the CPU time is greater than 100 to perform some action. The following query is evaluated to true when it returns some result. That means, when the mbean instrumenting the value of the ProcessCpuTime attribute of the operating system MBean, has an attribute greater than 100.

The first step, selects MBeans whose type property equals OperatingSystem. Then, the second step selects the ProcessCpuTime attribute and the last predicate filters the result for a value greater than 100.

```
$d/mbean::*[mbtype(.)=='OperatingSystem']/@ProcessCpuTime[value(.)>100]
```

Comparison with JMX Java API

Listing code below (figure 4) shows the Java code equivalent to the second query using the JMX API. As we can see, FPath has not only a more readable syntax but it is also more compact than a general purpose language like Java and its API (JMX API here).



```
try{
    String urlString =
        "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jrmpconnector_jonas";
    JMXServiceURL url = new JMXServiceURL(urlString);

    JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
    MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();

    //Selects MBeans with domain joramClient and any name
    ObjectName objectName = new ObjectName("joramClient:type=queue,*");

    //Selects MBeans with attribute PendingMessages greater than 5
    QueryExp queryExp = Query.gt(Query.attr("PendingMessages"),Query.value(5));

    //Runs the query over the server
    Set<ObjectInstance> objs = mbsc.queryMBeans(objectName, queryExp);

    //Evaluates the result
    for (ObjectInstance obj : objs) {
        System.out.println("Object = " + obj.getObjectName());
    }
}catch(Exception ex){
    System.out.println("Ooops!");
    ex.printStackTrace();
}
```

figure 4. Sample Java code showing the use of JMX API

4 Enhanced design and administration facilities

4.1 A generic software engineering approach with TUNE

In this section, we describe the results from our investigations in designing higher level paradigms for the specification of autonomic management policies. By management policies, we mean any input that has to be defined by an administrator in order to describe the legacy software to encapsulate, the software architecture to deploy and the reconfiguration to perform at runtime.

In a previous deliverable (SP1-L2) we have already introduced the design of a Wrapping Description Language for the encapsulation of legacy software in well formed Fractal components. This work goes a step further and aims at providing language support for simplifying the different administration tasks (wrapping, deployment, configuration, launching, reconfiguration).

4.1.1 Rationale

Component-based management

The basic idea underlying the Jasmine/Jade autonomic administration system is to encapsulate the managed elements (legacy software) in software components and to administrate the environment as a component architecture. We refer to this approach as the component-based management approach.

Component-based management aims at providing a uniform view of a software environment composed of different types of servers. Each managed server is encapsulated in a component and the software environment is abstracted as a component architecture. Therefore, deploying, configuring and reconfiguring the software environment is achieved by using the tools associated with the used component-based middleware. This solution is followed by several research projects, including Jasmine. In the current platform, an administrator can wrap legacy software in components (Jasmine/Jade relies on the Fractal component model), describe a software environment to deploy using the component model ADL (Architecture Description Language) and implement reconfiguration programs (autonomic managers) using the component model's interfaces (Java interfaces in Fractal).

Any software managed with Jade is wrapped in a Fractal component which interfaces its administration procedures. Therefore, the Fractal component model is used to implement a management layer (figure 5) on top of the legacy layer (composed of the actual managed software). In the management layer, all components provide a management interface for the encapsulated software, and the corresponding implementation (the wrapper) is specific to each software (e.g. the Apache web server in the case of J2EE). Fractal's control interfaces allow managing the element's attributes and bindings with other elements, and the management interface of each component allows controlling its internal configuration state. Relying on this management layer, sophisticated administration programs can be implemented, without having to deal with complex, proprietary configuration interfaces, which are hidden in the wrappers.

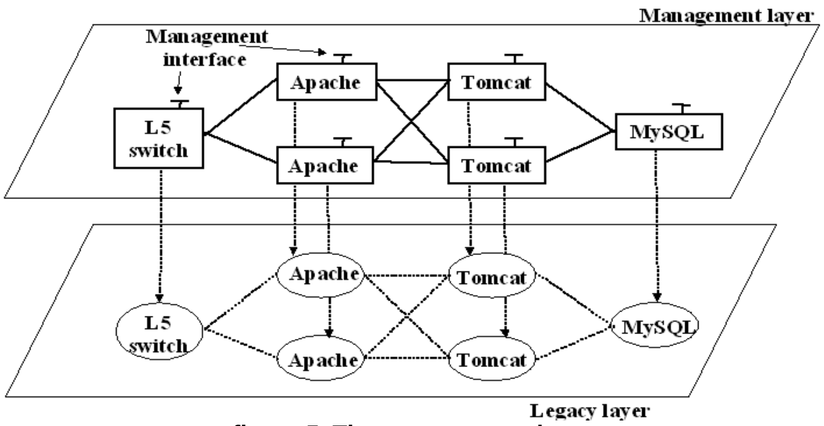


figure 5. The management layer

Here, we distinguish two important roles:

- the role of the management and control interfaces is to provide a means for configuring components and bindings between components. It includes methods for navigating in the component-based management layer or modifying it to implement reconfigurations.
- the role of the wrappers is to reflect changes in the management layer onto the legacy layer. The implementation of a wrapper for a specific software may also have to navigate in the component management layer, to access key attributes of the components and generate legacy software configuration files. For instance, the configuration of an Apache server requires to know the name and location of the Tomcat servers it is bound to.

Motivations

Component-based autonomic computing has proved to be a very convenient approach. The experiments we conducted with Jade for managing J2EE (and other) infrastructures validated this design choice. But as Jade was used by external users, we observed that:

- wrapping components are difficult to implement. The developer needs to have a good understanding of the component model we use (Fractal),
- deployment is not very easy. ADLs are generally very verbose and still require a good understanding of the underlying component model. Moreover, if we consider large scale software infrastructure such as those deployed over a grid, deploying a thousand of servers requires an ADL deployment description file of several thousands of lines,
- autonomic managers (reconfiguration policies) are difficult to implement as they have to be programmed using the management and control interfaces of the management layer. This also required a strong expertise regarding the used component model.

All these observations led us to the conclusion that a higher level interface was required for describing the encapsulation of software in components, the deployment of a software environment potentially in large scale and the reconfiguration policies to be applied autonomically. We present our proposal in the next section.

4.1.2 Tune's management interface

As previously motivated, our goal is to provide a high level interface for the description of the application to wrap, deploy and reconfigure. This led us to the following design choices:

- Regarding wrapping, our approach is to introduce a Wrapping Description Language which is used to specify the behavior of wrappers. A WDL specification is interpreted by a generic wrapper Fractal component, the specification and the interpreter implementing an equivalent wrapper. Therefore, an administrator doesn't have to program any implementation of Fractal component.

- Regarding deployment, our approach is to introduce a UML profile for graphically describing deployment schemas. First, a UML based graphical description of such a schema is much more intuitive than an ADL specification, as it doesn't require expertise of the underlying component model. Second, the introduced deployment schema is more abstract than the previous ADL specification, as it describes the general organization of the deployment (types of software to deploy, interconnection pattern) in intension, instead of describing in extension all the software instances that have to be deployed. This is particularly interesting for applications like Diet where thousands of servers have to be deployed.
- Regarding reconfiguration, our approach is to introduce a UML profile for the description of state diagrams. These state diagrams are used to define workflows of operations that have to be performed for reconfiguring the managed environment. One of the main advantage of this approach, besides simplicity, is that state diagrams manipulate the entities described in the deployment schema and reconfigurations can only produce an (concrete) architecture which conforms with the abstract schema, thus enforcing reconfiguration correctness.

We respectively detail these three aspects.

A UML profile for deployment schemas

The UML profile we introduce for specifying deployment schemas is illustrated in figure 6 where a deployment schema is defined for a J2EE organization. A deployment schema describes the overall organization of a software infrastructure to be deployed. At deployment time, the schema is interpreted to deploy a component architecture. Each element (the boxes) corresponds to a software which can be instantiated in several component replicas. A link between two elements generates bindings between the components instantiated from these elements. Each binding between two components is bi-directional (actually implemented by 2 bindings in opposite directions), which allows navigation in the component architecture. An element includes a set of configuration attributes for the software (all of type String). Most of these attributes are specific to the software, but few attributes are predefined by Tune and used for deployment:

- **wrapper** is an attribute which gives the name of the WDL description of the wrapper,
- **legacyFile** is an attribute which gives the archive which contains the legacy software binaries and configuration files,
- **hostFamily** is an attribute which gives a hint regarding the dynamic allocation of the nodes where the software should be deployed,
- **initial** is an attribute which gives the number of instances which should be deployed. The default value is 1.

The schema in figure 6 describes a J2EE cluster containing one Apache, two Tomcats, one C-JDBC and two Mysql that should be deployed. A probe is linked with tomcat, which monitors the server in order to trigger a repair/reconfigure procedure. In this schema, a cardinality is associated with each link. If A(n) and B(m) are two linked elements in a schema, with an initial attribute (initial number of instances) n for A and m for B, the semantic of the cardinality is the following. A link A*(n)* *t-u* B*(m)* means that each A component should be bound with u B components and each B component should be bound with t A components. The cardinality is constrained by m*t=n*u with m>=u and n>=t.

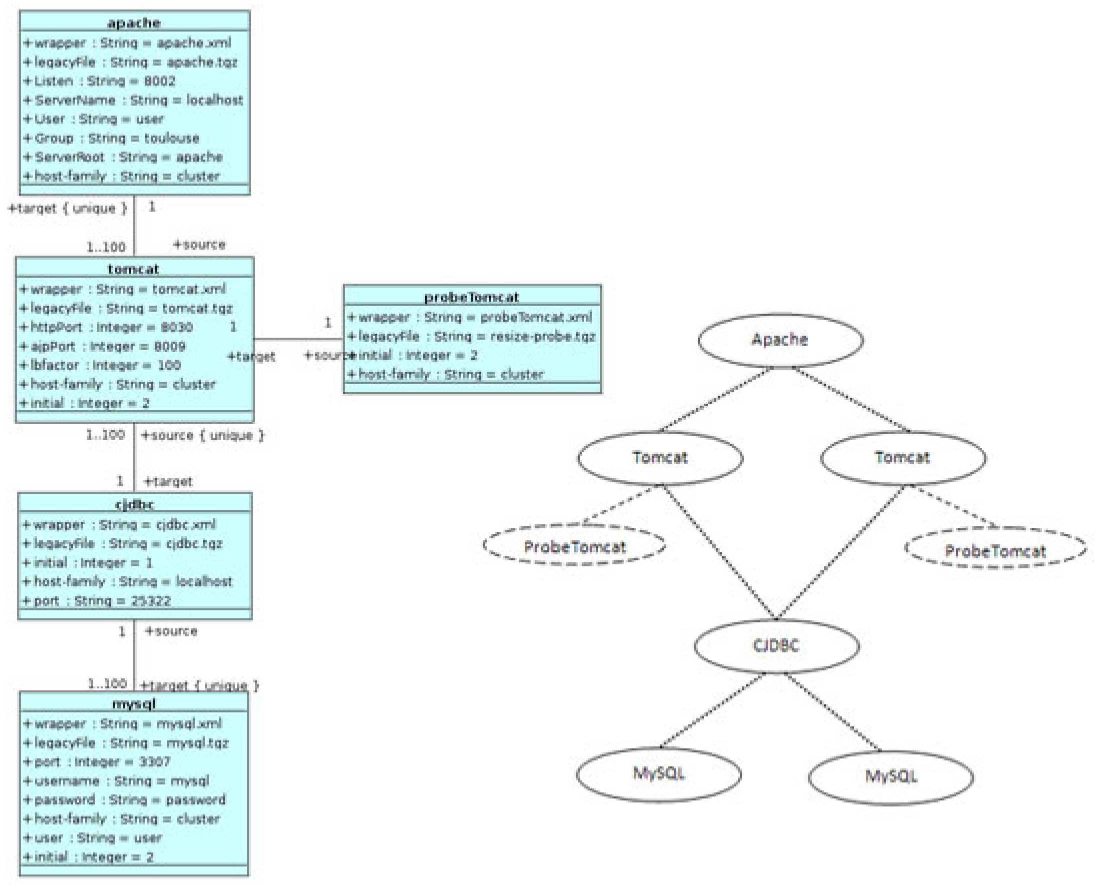

apache
+wrapper : String = apache.xml
+legacyFile : String = apache.tgz
+Listen : String = 8002
+ServerName : String = localhost
+User : String = user
+Group : String = toulouse
+ServerRoot : String = apache
+host-family : String = cluster
+target { unique }
1
1..100
+source
tomcat
+wrapper : String = tomcat.xml
+legacyFile : String = tomcat.tgz
+httpPort : Integer = 8030
+ajpPort : Integer = 8009
+lbfactor : Integer = 100
+host-family : String = cluster
+initial : Integer = 2
1
+target
1
+source
probeTomcat
+wrapper : String = probeTomcat.xml
+legacyFile : String = resize-probe.tgz
+initial : Integer = 2
+host-family : String = cluster
1..100
+source { unique }
1
+target
cjdbc
+wrapper : String = cjdbc.xml
+legacyFile : String = cjdbc.tgz
+initial : Integer = 1
+host-family : String = localhost
+port : String = 25322
1
+source
1..100
+target { unique }
mysql
+wrapper : String = mysql.xml
+legacyFile : String = mysql.tgz
+port : Integer = 3307
+username : String = mysql
+password : String = password
+host-family : String = cluster
+user : String = user
+initial : Integer = 2
Apache
Tomcat
Tomcat
ProbeTomcat
ProbeTomcat
CJDBC
MySQL
MySQL

figure 6. Deployment schema for a J2EE architecture

A Wrapping Description Language

Upon deployment, the above schema is parsed and for each element, a number of Fractal components are created. These Fractal components implement the wrappers for the deployed software, which provide control over the software. Each wrapper Fractal component is an instance of a generic wrapper which is actually an interpreter of a WDL specification.

A WDL description defines a set of methods that can be invoked to configure or reconfigure the wrapped software. The workflow of methods that have to be invoked in order to configure and reconfigure the overall software environment is defined thanks to an interface introduced in the next section.

Generally, a WDL specification provides start and stop operations for controlling the activity of the software, and a configure operation to reflect the values of the attributes (defined in the UML deployment schema) in the configuration files of the software. Notice that the values of these attributes can be modified dynamically. Other operations can be defined according to the specific management requirements of the wrapped software, these methods being implemented in Java.

The main motivations for the introduction of WDL are:

- to hide the complexity of the underlying component model (Fractal),
- That most of the needs should be met with a finite set of generic methods (that can be therefore reused).

```xml
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='apache'>
  <method name="start" key="extension.GenericStart" method="start_with_pid_linux" >
    <param value="$dirLocal/apache/bin/apachectl start" />
    <param value="LD_LIBRARY_PATH=$dirLocal" />
  </method>

  <method name="configure" key="extension.GenericConfigurePlainText" method="configure">
    <param value="$dirLocal/apache/conf/httpd.conf" />
    <param value=" "/>
    <param value="User:$User" />
    <param value="Group:$Group" />
    <param value="Listen:$Listen" />
    <param value="ServerName:$ServerName" />
    <param value="ServerRoot:$dirLocal/$ServerRoot" />
  </method>

  <method name="addWorkers" key="extension.AddTomcatWorker" method="addWorkers">
    <param value="$dirLocal/apache/conf/workers.properties" />
    <param value="name:$tomcat.srname" />
    <param value="type:ajp13" />
    <param value="host:$tomcat.nodeName" />
    <param value="port:$tomcat.ajpPort" />
    <param value="lbfactor:$tomcat.lbfactor" />
  </method>

  <method name="stop" key="extension.GenericStart" method="start_with_pid_linux" >
    <param value="$dirLocal/apache/bin/apachectl stop" />
    <param value="LD_LIBRARY_PATH=$dirLocal" />
  </method>
</wrapper>
```

figure 7. Apache WDL specification

The XML file in figure 7 shows an example of WDL specification which wraps an Apache computing server in a J2EE architecture. It defines *start* and *stop* methods which can be invoked to launch/stop the deployed Apache server, and a *configure* method which reflects configuration attributes in the configuration file of the Apache server, and an *addWorkers* method that adds the list of Tomcats to the Apache workers file. The Java implementations of these methods are generic and have been used in the wrappers of most of the software we wrapped (currently we have 2 implementations: one for XML configuration files like Tomcat configuration file, and another for plain text files like Apache configuration file). A method definition includes the description of the parameters that should be passed when the method is invoked. These parameters may be String constants, attribute values or combination of both (String expressions). All the attributes defined in the deployment schema can be used to pass the configured attributes as parameters of the method invocations. However, some additional attributes are automatically added and managed by Tune:

- *dirLocal* is the directory where the software is actually deployed on the target machine,
- *srName* is a unique name associated with the deployed component instance.

In figure 7, the *start* method takes as parameters the shell command that launch the server, and the environment variables that should be set:

- “$dirLocal/apache/bin/apachectl start” is the shell command that launches the server,
- “LD_LIBRARY_PATH=$dirLocal” is an environment variable to pass to the binary.

The *configure* method is implemented by the *GenericConfigurePlainText* Java class. This configuration method generates a configuration file composed of <attribute,value> pairs:

- “$dirLocal/apache/conf/httpd.conf” is the name of the configuration file to generate,
- “ ” is the separator between each attribute and value,
- and the attributes and value are separated by a ":" character.

It is sometimes necessary to navigate in the deployed component architecture to access key attributes of the components in order to configure the software. For instance, the configuration of an apache server requires knowing the name and location of the tomcat servers it is bound to. Therefore, in the Apache wrapper (figure 7), we need to access Tomcats parameters in order to set these hosts and ports variables. Since in the deployment schema there is a link between the Apache and Tomcat elements, there are bindings between the Apache and the Tomcats servers at the component level. These bindings allow navigating in the management layer. “$tomcat.srname” returns the list of the names of the Tomcats the Apache is bound with.

A UML profile for (re)configuration procedures

Reconfigurations are triggered by events. An event can be generated by a specific monitoring component (e.g. probes in the deployment schema) or by a wrapped legacy software which already includes its own monitoring functions.

Whenever a wrapper component is instantiated, a communication pipe is created (typically a UNIX pipe) that can be used by the wrapped legacy software to generate an event, following a specified syntax which allows for parameter passing. Notice that the use of pipes allows any software (implemented in any language environment such as Java or C++) to generate events. An event generated in the pipe associated with the wrapper is transmitted to the administration node where it can trigger the execution of reconfiguration programs (in our current prototype, the administration code, which initiates deployment and reconfiguration, is executed on one administration node, while the administrated software is managed on distributed hosts). An event is defined as an event type, the name of the component which generated the event and eventually an argument (all of type String).

For the definition of reactions to events, we introduced a UML profile which allows specifying reconfiguration as state diagrams. Such a state diagram defines the workflow of operations that must be applied in reaction to an event.

An operation in a state diagram can assign an attribute or a set of attributes of components, or invokes a method or a set of methods of components. To designate the components on which the operations should be performed, the syntax of the operations in the state diagrams allows navigation in the component architecture, similarly to the wrapping language.

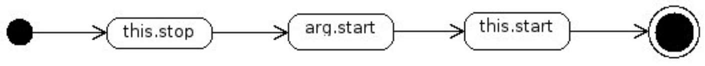


figure 8. State diagram for repair

For example, let's consider the diagram in figure 8 which is the reaction to a Tomcat failure. The event (fixTomcat) is generated by a probeTomcat component instance; therefore the variable “this” is the name of this probeTomcat component instance. Then:

- *this.stop* will invoke the stop method on the probing component (to prevent the generation of multiple events),
- *arg.start* will invoke the start method on the Tomcat component instance which is linked with the probe. This is the actual repair of the faulting Tomcat server,
- *this.start* will restart the probe associated with the Tomcat.

Notice that state diagram's operations are expressed using the elements defined in the deployment schema, and are applied on the actually deployed component architecture.

A similar diagram is used to start the deployed J2EE cluster, as illustrated in figure 9. In this diagram, when an expression starts with the name of an element in the deployment schema (apache or tomcat ...), the semantic is to consider all the instances of the element, which may result in multiple method invocations. The starting diagram ensures that (1) configuration files must be generated, and then (2) the servers must be started following the order.

Similar diagrams can be drawn to define the actions or methods that should be invoked while upsizing or downsizing a component in reaction to events of load peak.

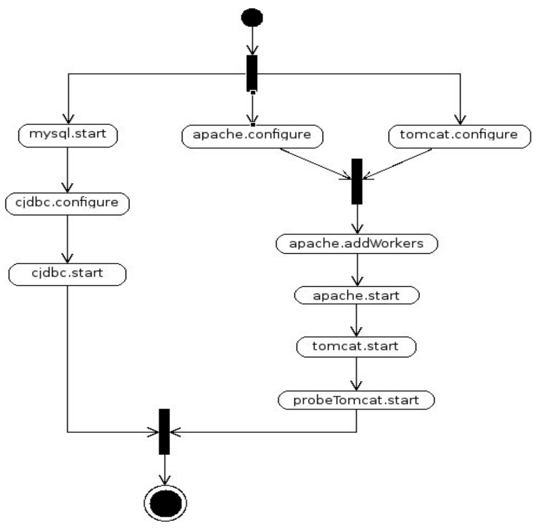


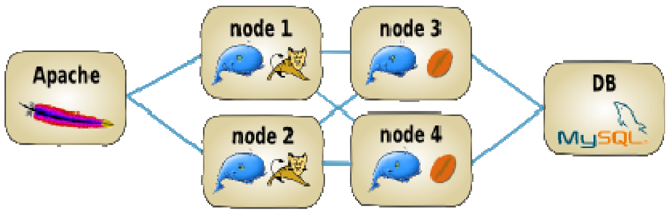
figure 9. State diagram for start

4.1.3 Conclusions

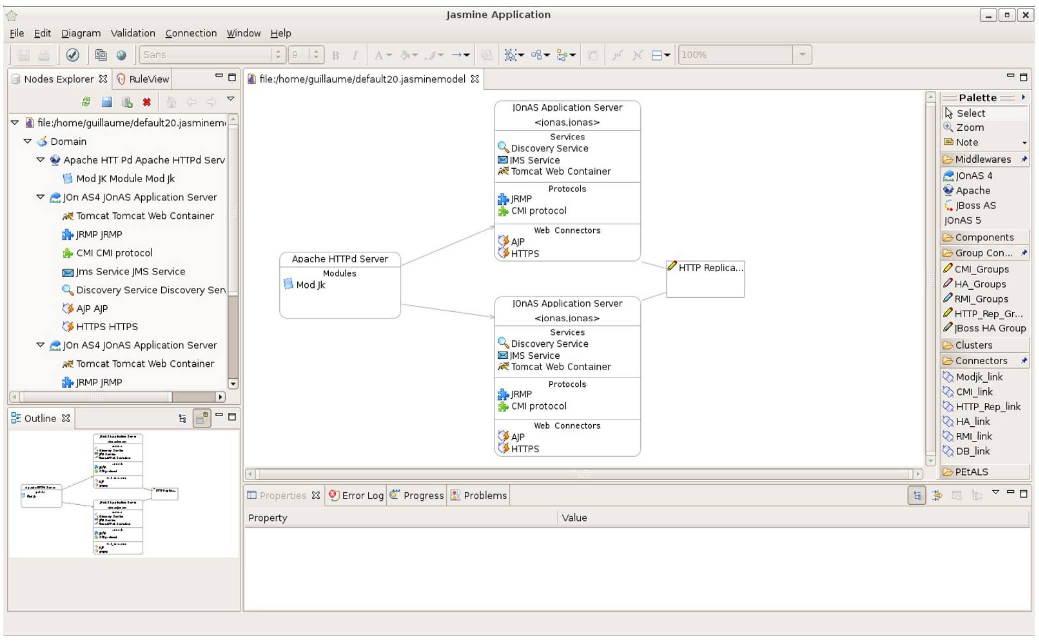
As computing environments are becoming increasingly sophisticated, there is a need for advanced system services to keep these platforms maintainable, i.e. providing tools that ease the deployment and administration of these distributed systems. With Tune, we propose a higher level interface for describing the encapsulation of software in components, the deployment of a software environment and the reconfiguration policies to be applied autonomically. This management interface is mainly based on UML profiles for the description of deployment schemas and the description of reconfiguration state diagrams. A tool for the description of wrapper is also introduced to hide the details of the underlying component model.

4.2 J2EE tooling with JASMINe Design

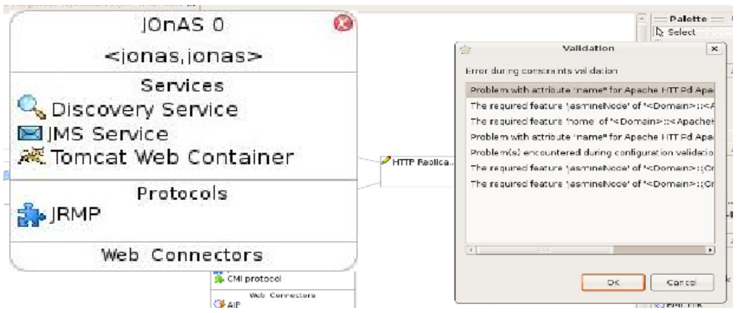
JASMINe Design aims at providing a high level interface for describing graphically a distributed middleware configuration. In a first time, the JOnAS's Java EE middleware configuration is addressed. Typically you may define a JOnAS cluster configuration with an Apache frontal, some web level instances, some ejb level instances and a database.



JASMINe Design relies on a Eclipse EMF/GMF graphical interface. The GUI provides all JOnAS cluster elements as graphical icons that the user can drag and drop to design a configuration, as shown in the screen shot below.



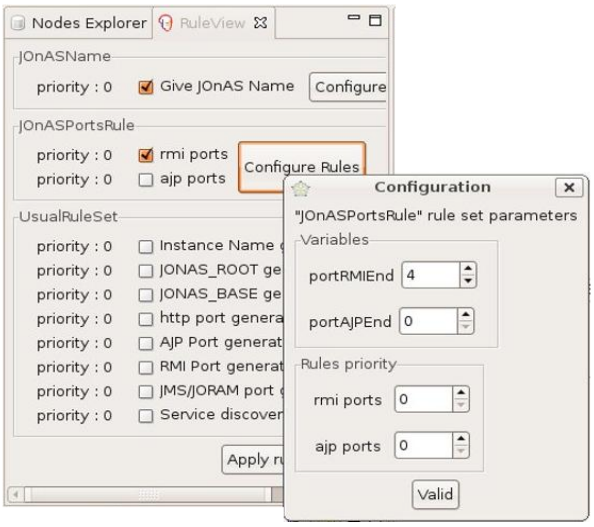
The tool provides a system for validating the middleware configuration before deployment. It enables to detect some misses in the configuration description but also some incompatibilities. For example, if a mandatory parameter is not set by the user, the validation system will notify the operator through an error message and/or an error highlighting in the graphic element. Regarding the incompatibilities, the tool is able to detect some inconsistent links between middleware elements such as a connection between an Apache front end with an EJB container. The checking is implemented through a set of OCL constraints applied to the EMF model and a set of rules implemented with the Drools rules engine. The user may extend the built-in rules for adding its own validation rules. For example, each enterprise can define its own value domain for a middleware parameter with a customized rule for controlling that.



Error during constraints validation

Problem with attribute 'name" for Apache HTTPd Apa
The required feature 'jasmineNode' of '<Domain>::<A
The required feature 'home' of '<Domain>::<ApacheH
Problem with attribute 'name" for Apache HTTPd Apa
Problem(s) encountered during configuration validatio
The required feature 'jasmineNode' of '<Domain>::Cr
The required feature 'jasmineNode' of '<Domain>::Cr

By the way, JASMINe Design provides a wizard for managing automatic configuration rules, Drools rules engine based as well. Indeed, each organization has its owns administration policy that defines how the middleware must be installed and configured. A such wizard enforces the policy adoption by delegating a part of the middleware setting to the tool. Human error are reduced and the maintenance is easier since the different configurations are homogeneous within the company. A first example of rule is the port numbers allocation across the infrastructure. The operator can specify a port range for each network interface and the tool is getting in when configuring a middleware element. Another example is the JOnAS instance name building which may be relevant to automate for ensuring the adoption of the rules naming into the enterprise. And one may have the same requirement for defining the middleware's home directory path name.


Nodes Explorer | RuleView

JOnASName
priority : 0 ☑ Give JOnAS Name [Configure]

JOnASPortsRule
priority : 0 ☑ rmi ports [Configure Rules]
priority : 0 ☐ ajp ports

UsualRuleSet
priority : 0 ☐ Instance Name
priority : 0 ☐ JONAS_ROOT ge
priority : 0 ☐ JONAS_BASE ge
priority : 0 ☐ http port genera
priority : 0 ☐ AJP Port generat
priority : 0 ☐ RMI Port generat
priority : 0 ☐ JMS/JORAM port
priority : 0 ☐ Service discover

[Apply r

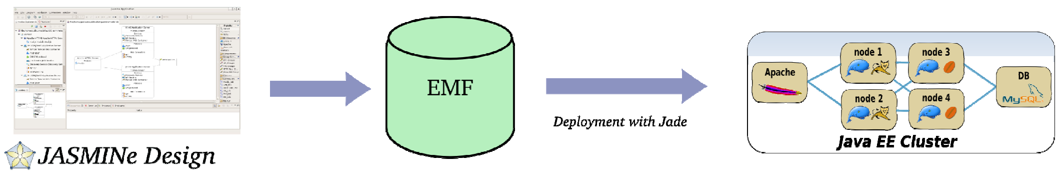
Configuration
"JOnASPortsRule" rule set parameters

Variables
portRMIEnd 4
portAJPEnd 0

Rules priority
rmi ports 0
ajp ports 0

[Valid]

Once the user has finished to describe the middleware configuration, he can use the Jade framework for deploying it over the infrastructure.

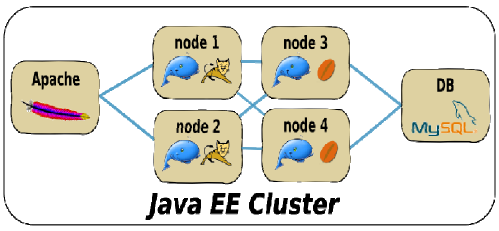


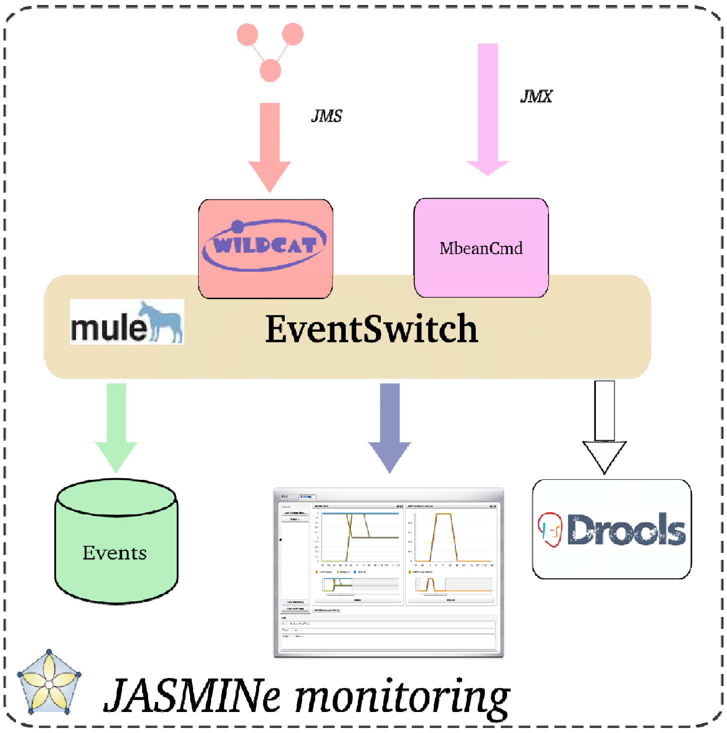
Thanks to the graphical interface and the Model Driven Configuration principle, the distributed middleware configuration is simpler and quicker for the user who may devote more time to the architecture design. The validation and automatic configuration features do reduce the risk of errors at the deployment time and promote the middleware dissemination within the enterprise.

4.3 Monitoring support with MbeanCmd and the EventSwitch

JASMINe monitoring provides an infrastructure for supervizing a middleware distributed configuration. It aims at offering two main features: performance tracking and error detection. The infrastructure is composed of:

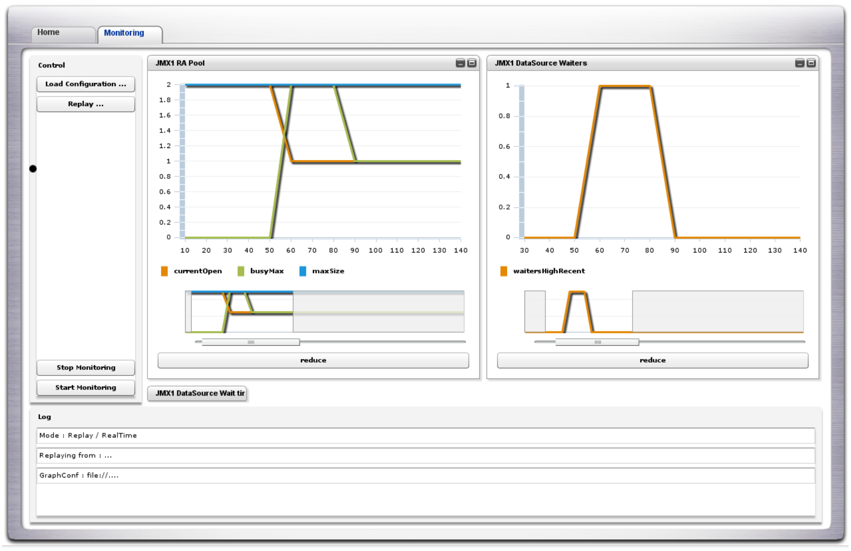
- An EventSwitch element based on the Mule's lightweight ESB in charge of routing the monitoring events from the events producer (probes, wildcat) towards the events consumers (persistent storage, console, rules engine).The EventSwitch ensures the extensibility of the architecture, any additional producer or consumer may be plugged without a big impact.
- A set of JMX probes, named *MBeanCmd*, for collecting monitoring data through a JMX interface. Some built-in options are provided for getting some well-known application server indicators such as the transaction throughput or the current HTTP session number. An embedded deployment mode into the EventSwitch coupled with a JMX connection pool does ensure the scalability when monitoring a whole Java EE cluster.
- An optional distributed events mediation infrastructure, named Wildcat, enabling to collect, gather, aggregate, filter the monitoring events composed of. Wildcat is described in detail in a dedicated section of this document.
- An events persistent storage system with an EJB3/JPA interface and an event database.
- A Web 2.0 console (named EoS console) enabling to analyze the performance into graphics.
- A rules engine providing to the user the capacity to implement its own policy administration rules for detecting errors. Any kind of action may be implemented in Java language for notifying the operator when an error occurs (mail sending, log message, and so on).





MBeanCmd tool allows to very easily get information from available Mbeans. This tool is available as a Java command (mbean.jar), thus enabling scripting, it relies on the JMX Remote interface and provides the capability to get and set MBean attributes, to invoke MBean methods. It is used for very easily probing the most relevant JOnAS indicators like transactions, data sources, http connectors, threads pools, jms statistics, etc. JVM Mbeans can be polled as well for getting the current cpu load or the memory usage. MBeanCmd can be used both in a standalone mode or embedded in the EventSwitch. In a standalone mode, it allows file storage (CSV), provides a replay mode and a graphic swing console.

EoS console (Eye of SOA) enables to track the monitoring events into graphics. Input events can be listened from the EventSwitch through a JMS topic or retrieved from the events database. Several graphics can be configured at the same time with different curves and scales. The interface is dynamic (module loaded on demand, self-sizing windows) and intuitive (adjustable zoom , automatic rescaling). An export function on the graphs into JPEG pictures enables to put the results into a report.



```
Mode : Replay / RealTime
Replaying from : ...
GraphConf : file://....
```

The Drools rules engine enables to implement some high level error detection rules. For example, the user can define a threshold and a notification action when an indicator reaches the threshold. Actions are written in Java and can use the Java EE services of the underlying application server (mail, Web Service invocation, database access) as JASMINe monitoring is a Java EE application. Below, a message is printed in the console when the cpu load reaches 80%.

```
rule "Your First Rule"
        when
                $event : JasmineEventEB(

                           probe == "cpu:load",

                           $value : value,

                           Integer.parseInt((String) value)) > 80)

        then
                #actions
                System.out.println("Cpu load reaches a threshold");
                retract ($event);

end
```

JASMINe monitoring improves the reactivity when an error occurs and thus contributes to reduce the administration costs and the continuity of service. JASMINe monitoring is an element of the SOA governance and gives to the operator an overview of the system behavior, health, and efficiency.

4.4 Shelbie and FPath JMX

Shelbie is a dynamic, modular and extensible command line application written in Java and embeds in the JOnAS 5 application server but which may be used in others context as well. It provides features close to those provided by the BASH interpreter.

For users, Shelbie is an easy to use command line interface which provides useful features like command completion or history. Shelbie enables user to write scripts for automating complex tasks and is remotely accessible thanks to an embedded SSH server.

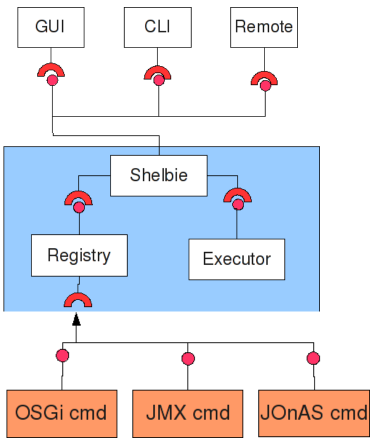
For developers, Shelbie is an easy to extend Shell which provides facilities to develop commands. One goal of Shelbie is to help developer to focus on application logic and not in painful issues like command option processing. Shelbie offers tools to customize command rendering including text coloration and decoration.

More than a simple command line application, Shelbie can be considered as a framework which helps building rich command line application. Shelbie is portable and works in commons operating system (Linux, MacOSX, Windows).

Main features are:

- Rich line editing including tab-completion, persistent history and user input masking (Jline [http://jline.sourceforge.net/]).
- Remote access using embedded SSH2 server (Jaramiko) [http://www.lag.net/jaramiko/]
- Annotation-based command line parsing (Args4j) [https://args4j.dev.java.net/]
- Scripting support using the powerful Groovy [http://groovy.codehaus.org/] language
- Support ANSI text coloration and decoration
- Dynamic module discovery using OSGi [http://www.osgi.org/Main/HomePage] and iPOJO [http://felix.apache.org/site/ipojo.html]
- Detailed help on command

Three sets of commands have been delivered in a first version as shown in the figure:



- a set of OSGi commands for controlling the Felix framework (bundle install, start, stop)
- A set of JOnAS commands for controlling the application server (start, stop, list of JNDI entries, application deployment, ...)
- a set of JMX commands relying on the FPath/JMX framework. The FPath/JMX project is detailed in another section of this document. It provides a domain specific language (DSL) for navigating in the MBeans structure and retrieving one or several MBeans attributes.

Example of syntax for a JMX command on top of FPath/JMX:

```
//get a list of the JORAM queues
joramClient/mbean::*[mbtype(.)=='queue']

//get the number of messages in the queue 'sampleQueue'
joramClient/mbean::*[mbname(.)=='sampleQueue']/@PendingMessages
```

5 Conclusion

The second release of the Selfware platform's services provides higher-level tools and APIs, mostly based on the underlying Fractal component-based architecture. Monitoring concerns and Java technologies have been mainly addressed with WildCATv2, JASMINe Design, Shelbie, FPathJMX and MBeanCmd. TUNE is different in that it takes a pure software engineering approach, and is finally independent from the target programming languages and models.

We notice that some distance with the Fractal architecture is now provided, in order to support new facilities that are not going to be handled by Fractal-aware operators, administrators or autonomic policies designers. Moreover, some of these tools turn to be independent from Fractal, and may apply to less specific environments, mostly (but not only) Java-based. This observation is just illustrating the incremental way of building the Selfware platform, first focusing on a sound architectural substrate supporting a well-controlled runtime management, and then going towards supporting third party actors that will finally need to use and customize a higher-level framework to design and administrate autonomic systems.

Possible further work about the Selfware services should address the following issues:

- monitoring requirements have resulted in a great number of tools, and some of them should be unified, in order to grant both the component-based architectural approach (as provided by Composite Probes) and friendly high-level APIs (as provided by WildCAT).
- decision making is the other hot topic that deserves some extra work. Here again, unifying the architectural approach (as provided by the ECA rules framework) with user-friendly high-level policy tools (such as Drools rule engine used with JASMINe) would enable a better integration and manageability within the Selfware platform.
- an overall review of the Selfware services should also be carried out with respect to its applicability to typical real systems, with regard to scalability and complexity. One of the key issues is how to deal with heterogeneous control loops: for example, a self-optimization control loop may have to be conciliated with a self-protection loop, or a performance self-optimization loop may have to be conciliated with an electrical power self-optimization loop. Then, simple rule-based decision making might not be practically effective, and more advanced reasoning or learning mechanisms may be necessary. Of course, the scalability issue also tackles the monitoring service, since a huge number of events are to be generated, transported and queried in a widely distributed system.