AUTHORIZED DOCUMENTATION

Administrator Reference

Novell®
PlateSpin® Orchestrate

2.0

February 23, 2008

www.novell.com

Legal Notices

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.
www.novell.com

Online Documentation: To access the latest online documentation for this and other Novell products, see the Novell Documentation Web page (http://www.novell.com/documentation).

Novell Trademarks

Third-Party Materials

Contents

About This Guide

This *Administration Guide* introduces the processes you can use with PlateSpin® Orchestrate 2.0 from Novell®, including the applied use of the PlateSpin Orchestrate Development Client and various command line tools. The guide provides an introductory overview of PlateSpin Orchestrate and explains how it administers and manages work on the resources of the data center. The guide is organized as follows:

- Chapter 1, "Basic PlateSpin Orchestrate Concepts," on page 9
- Appendix A, "PlateSpin Orchestrate Security," on page 27
- Appendix B, "Server Discovery and Multicasting," on page 31

For reference information about the Orchestrate Server or the Orchestrate Development Client, see the *PlateSpin Orchestrate 2.0 Development Client Reference*. For information about the Orchestrate Command Line Tools, see *PlateSpin Orchestrate 2.0 Command Line Reference*.

Audience

This book is intended for data center managers and IT or Operations administrators. It assumes that users of the product have the following background:

- General understanding of network operating environments and systems architecture.
- Knowledge of basic UNIX* shell commands and text editors.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation, or go to www.novell.com/documentation/feedback.html (http://www.novell.com/documentation/feedback.html) and enter your comments there.

Documentation Updates

For the most recent version of this *Administration Guide*, visit the PlateSpin Orchestrate Web site (http://www.novell.com/documentation/pso_orchestrate20/).

Documentation Conventions

In Novell documentation, a greater-than symbol (>) is used to separate actions within a step and items in a cross-reference path.

A trademark symbol (®, ™, etc.) denotes a Novell trademark. An asterisk (*) denotes a third-party trademark.

When a single pathname can be written with a backslash for some platforms or a forward slash for other platforms, the pathname is presented with a backslash. Users of platforms that require a forward slash, such as Linux* or UNIX, should use forward slashes as required by your software.

Basic PlateSpin Orchestrate Concepts

1

This section contains the followings information:

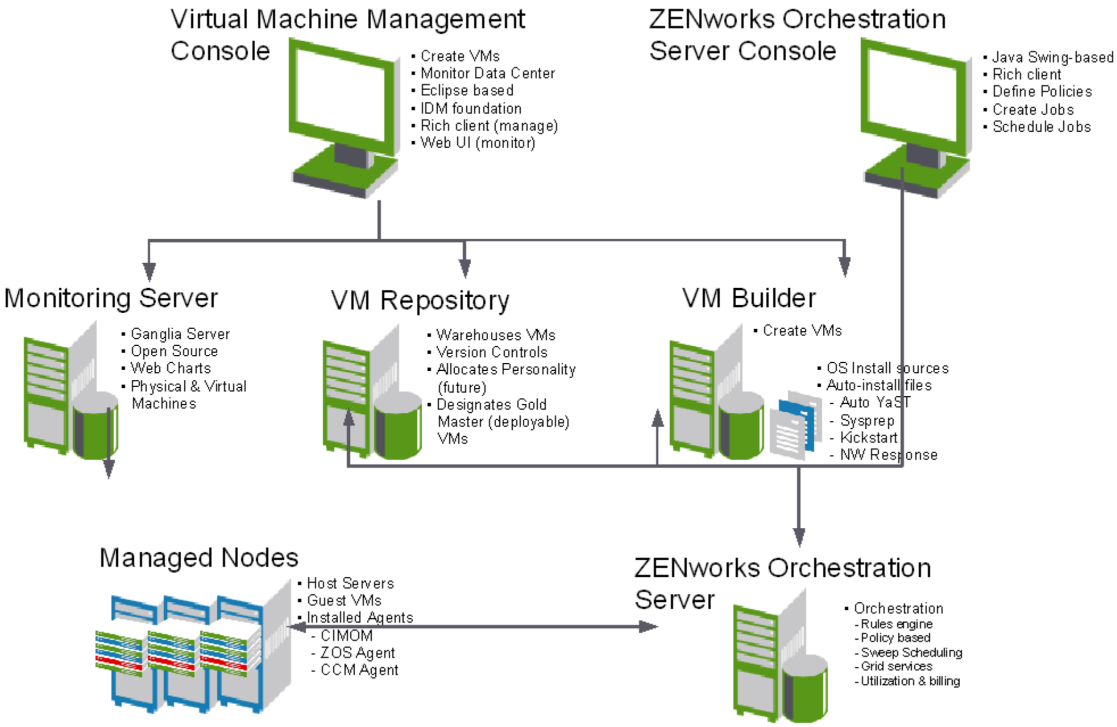
- Section 1.1, “Understanding PlateSpin Orchestrate Architecture,” on page 9
- Section 1.2, “Understanding PlateSpin Orchestrate Functionality,” on page 20

1.1 Understanding PlateSpin Orchestrate Architecture

PlateSpin Orchestrate from Novell is an advanced datacenter management solution designed to manage all network resources. It provides the infrastructure that manages group of ten, one hundred, or thousands of physical or virtual resources.

PlateSpin Orchestrate is equally apt at performing a number of distributed processing problems. From high performance computing, the breaking down of work into lots of small chunks that can be processed in parallel through distributed job scheduling. The following figure shows the product’s high-level architecture:

Figure 1-1 *PlateSpin Orchestrate Architecture*



This section contains information about the following topics:

- Section 1.1.1, “The PlateSpin Orchestrate Agent,” on page 10

- Section 1.1.2, “The Resource Monitor,” on page 11
- Section 1.1.3, “Entity Types and Managers,” on page 11
- Section 1.1.4, “Jobs,” on page 14
- Section 1.1.5, “Constraint-Based Job Scheduling,” on page 17
- Section 1.1.6, “Understanding PlateSpin Orchestrate API Interfaces,” on page 18

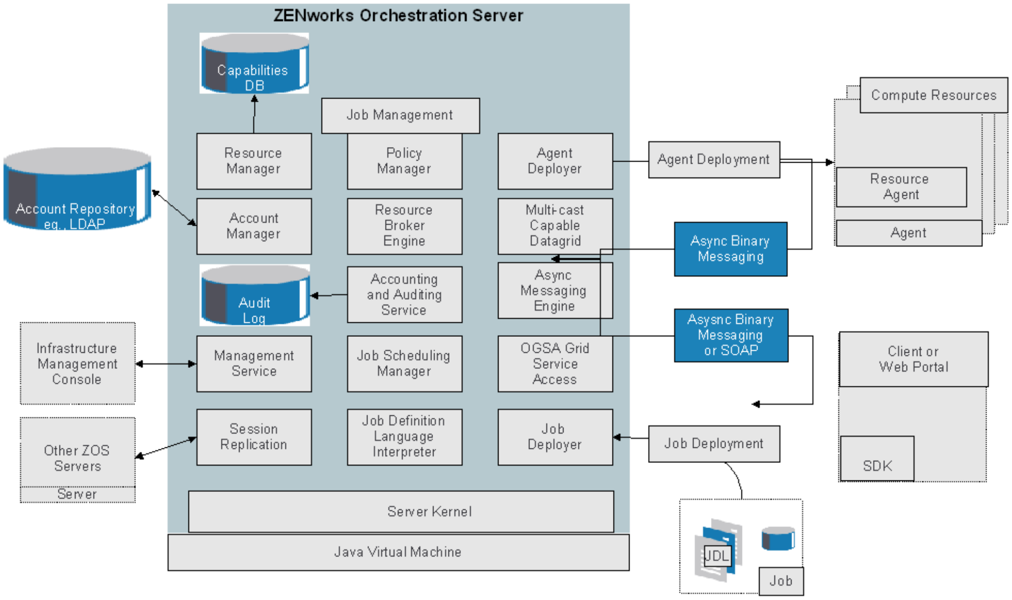
1.1.1 The PlateSpin Orchestrate Agent

Agents are installed on all managed resources as part of the product deployment. The agent connects every managed resource to its configured server and advertises to the PlateSpin Orchestrate Server that the resource is available for tasks. This persistent and auto-reestablishing connection is important because it provides a message bus for the distribution of work, collection of information about the resource, per-job messaging, health checks, and resource failover control.

After resources are enabled, PlateSpin Orchestrate can discover, access, and store detailed abstracted information—called “facts”—about every resource. Managed resources, referred to as “nodes,” are addressable members of the Orchestrate Server “grid” (also sometimes called the “matrix”). When integrated into the grid, nodes can be deployed, monitored, and managed by the Orchestrate Server, as discussed in Section 1.2, “Understanding PlateSpin Orchestrate Functionality,” on page 20.

An overview of the PlateSpin Orchestrate grid architecture is illustrated in the figure below, much of which is explained in this guide:

Figure 1-2 *PlateSpin Orchestrate Server Architecture*



For additional information about job architecture, see “Job Architecture” in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

1.1.2 The Resource Monitor

PlateSpin Orchestrate enables you to monitor your system computing resources using the built-in Resource Monitor. To open the Resource Monitor in the Development Client, see “Monitoring Server Resources” in the *PlateSpin Orchestrate Administration Guide*.

1.1.3 Entity Types and Managers

The following entities are some of key components involved in the Orchestrate Server:

- “Resources” on page 11
- “Users” on page 11
- “Job Definitions” on page 12
- “Job Instances” on page 12
- “Policies” on page 12
- “Facts” on page 12
- “Constraints” on page 13
- “Groups” on page 13
- “VM: Hosts, Images, and Instances” on page 13
- “Templates” on page 13

Resources

All managed resources, which are called nodes, have an agent with a socket connection to the Orchestrate Server. All resource use is metered, controlled, and audited by the Orchestrate Server. Policies govern the use of resources.

PlateSpin Orchestrate allocates resources by reacting as load is increased on a resource. As soon as we go above a threshold that was set in a policy, a new resource is allocated and consequently the load on that resource drops to an acceptable rate.

You can also write and jobs that perform cost accounting to account for the cost of a resource up through the job hierarchy, periodically, about every 20 seconds. For more information, see “Auditing and Accounting Jobs” in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

A collection of jobs, all under the same hierarchy, can cooperate with each other so that when one job offers to give up a resource it is reallocated to another similar priority job. Similarly, when a higher priority job becomes overloaded and is waiting on a resource, the system “steals” a resource from a lower priority job, thus increasing load on the low priority job and allocating it to the higher priority job. This process satisfies the policy, which specifies that a higher priority job must complete at the expense of a low priority job.

Users

PlateSpin Orchestrate users must authenticate to access the system. Access and use of system resources are governed by policies.

Job Definitions

A job definition is described in the embedded enhanced Python script that you create as a job developer. Each job instance runs a job that is defined by the Job Definition Language (JDL). Job definitions might also contain usage policies. For more information, see "Job Class" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

Job Instances

Jobs are instantiated at runtime from job definitions that inherit policies from the entire context of the job (such as users, job definitions, resources, or groups). For more information, see "JobInfo" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

Policies

Policies are XML documents that contain various constraints and static fact assignments that govern how jobs run in the PlateSpin Orchestrate environment.

Policies are used to enforce quotas, job queuing, resource restrictions, permissions, and other job parameters. Policies can be associated with any PlateSpin Orchestrate object. For more information, see Section 1.2.2, "Policy-Based Management," on page 21.

Facts

Facts represent the state of any object in the PlateSpin Orchestrate grid. They can be discovered through a job or they can be explicitly set.

Facts control the behavior a job (or joblet) when it's executing. Facts also detect and return information about that job in various UIs and server functions. For example, a job description that is set through its policy and has a specified value might do absolutely nothing except return immediately after network latency.

There are three basic types of facts:

- **Static:** Facts that require you to set a value. For example, in a policy, you might set a value to be False. Static facts can be modified through policies.
- **Dynamic:** Facts produced by the PlateSpin Orchestrate system itself. Policies cannot override dynamic facts. They are read only and their value is determined by the PlateSpin Orchestrate Server itself.
- **Computed:** Facts derived from a value, like that generated from the cell of a spreadsheet. Computed facts have some kind of logic behind them which derive their values.

  For example, you might have two numeric facts that you want expressed in another fact as an average of the two. You could compose a computed fact which averages two other facts and express it as an average value under a certain fact name. This enables you to create facts that represent other metrics on the system that are not necessarily available in the default set, or are not static to anything that might impact other dynamic facts.

For more information about facts, see "Facts" in "Policy Elements" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

Constraints

In order for the PlateSpin Orchestrate to choose resources for a job, it uses resource constraints. A resource constraint is some Boolean logic that executes against facts in the system. Based upon this evaluation, it will only consider resources that match the criteria that have been set up by use of constraints.

For more detailed information, see “Working with Facts and Constraints” in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference* and the following JDL constraint definitions listed in the same guide:

- “AndConstraint()”
- “BinaryConstraint”
- “Constraint”
- “ContainerConstraint”
- “ContainsConstraint”
- “DefinedConstraint”
- “EqConstraint”
- “GeConstraint”
- “GtConstraint”
- “LeConstraint”
- “LtConstraint”
- “NeConstraint”
- “NotConstraint”
- “OrConstraint”
- “UndefinedConstraint”

Groups

Resources, users, job definitions and virtual machines (VM) are managed in groups with group policies that are inherited by members of the group.

VM: Hosts, Images, and Instances

A virtual machine host is a resource that is able to run guest operating systems. Attributes (facts) associated with the VM host control its limitations and functionality within the Orchestrate Server. A VM image is a resource image that can be cloned and/or provisioned. A VM instance represents a running copy of a VM image.

Templates

Templates are images that are meant to be cloned (copied) prior to provisioning the new copy. For more information, see “Creating a Template from a VM” in the *PlateSpin Orchestrate 2.0 VM Client Guide and Reference*.

1.1.4 Jobs

The Orchestrate Server manages all nodes by administering jobs (and the functional control of jobs at the resource level by using joblets), which control the properties (facts) associated with every resource. In other words, jobs are units of functionality that dispatch data center tasks to resources on the network such as management, migration, monitoring, load balancing, etc.

PlateSpin Orchestrate provides a unique job development, debugging, and deployment environment that expands with the demands of growing data centers.

As a job developer, your task is to develop jobs to perform a wide array of work that can be deployed and managed by PlateSpin Orchestrate.

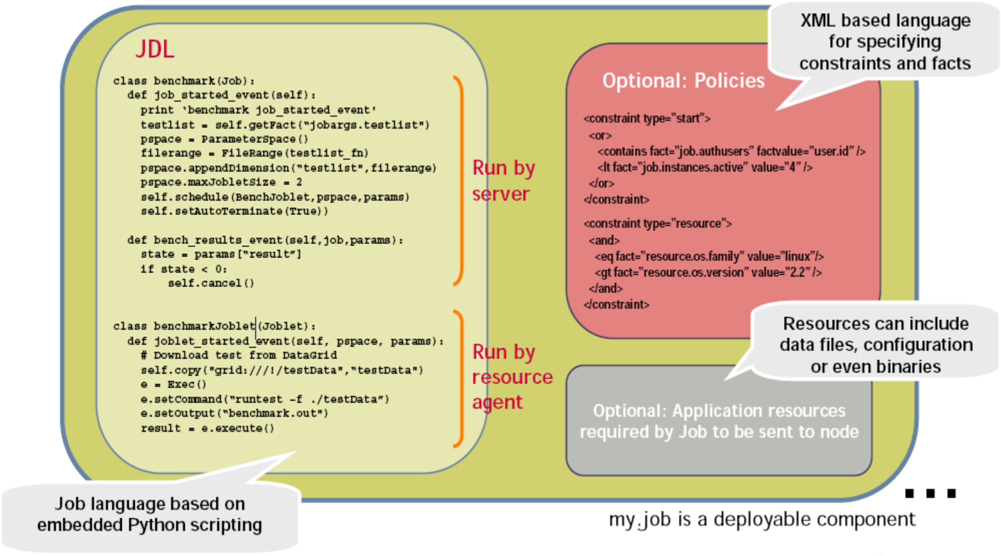
Jobs, which run on the Orchestrate Server, can provide functions within the PlateSpin Orchestrate environment that might last from seconds to months. Job and joblet code exist in the same script file and are identified by the `.jdl` extension. The `.jdl` script contains only one job definition and zero or more joblet definitions. A `.jdl` script can have only one Job subclass. As for naming conventions, the Job subclass name does not have to match the `.jdl` filename; however, the `.jdl` filename is the defined job name, so the `.jdl` filename must match the `.job` filename that contains the `.jdl` script. For example, the job files (`demoIterator.jdl` and `demoIterator.policy`) included in the `demoIterator` example job are packaged into the archive file named `demoIterator.job`, so in this case, the name of the job is `demoIterator`.

A job file also might have policies associated with it to define and control the job's behavior and to define certain constraints to restrict its execution. A `.jdl` script that is accompanied by a policy file is typically packaged in a job archive file (`.job`). Because a `.job` file is physically equivalent to a Java archive file (`.jar`), you can use the JDK JAR tool to create the job archive.

Multiple job archives can be delivered as a management pack in a service archive file (SAR) identified with the .sar extension. Typically, a group of related files are delivered this way. For example, the Xen30 management pack is a SAR.

As shown in the following illustration, jobs include all of the code, policy, and data elements necessary to execute specific, predetermined tasks administered either through the PlateSpin Orchestrate Development Client, or from the zos command line tool.

***Figure 1-3** Components of a Job (`my.job`,)*



```
class benchmark(Job):
  def job_started_event(self):
    print 'benchmark job_started_event'
    testlist = self.getFact("jobargs.testlist")
    pspace = ParameterSpace()
    filerange = FileRange(testlist_fn)
    pspace.appendDimension("testlist",filerange)
    pspace.maxJobletSize = 2
    self.schedule(BenchJoblet,pspace,params)
    self.setAutoTerminate(True))

  def bench_results_event(self,job,params):
    state = params["result"]
    if state < 0:
        self.cancel()


class benchmarkJoblet(Joblet):
  def joblet_started_event(self, pspace, params):
    # Download test from DataGrid
    self.copy("grid:///!/testData","testData")
    e = Exec()
    e.setCommand("runtest -f ./testData")
    e.setOutput("benchmark.out")
    result = e.execute()
```

```
<constraint type="start">
 <or>
   <contains fact="job.authusers" factvalue="user.id" />
   <lt fact="job.instances.active" value="4" />
 </or>
</constraint>

<constraint type="resource">
 <and>
  <eq fact="resource.os.family" value="linux"/>
  <gt fact="resource.os.version" value="2.2" />
 </and>
</constraint>
```

JDL
Run by server
Run by resource agent
Optional: Policies
XML based language for specifying constraints and facts
Resources can include data files, configuration or even binaries
Optional: Application resources required by Job to be sent to node
Job language based on embedded Python scripting
...
my.job is a deployable component

Because each job has specific, predefined elements, jobs can be scripted and delivered to any agent, which ultimately can lead to automating almost any datacenter task. Jobs provide the following functionality:

- "Controlling Process Flow" on page 15
- "Parallel Processing" on page 15
- "Managing the Cluster Life Cycle" on page 16
- "Discovery Jobs" on page 16
- "System Jobs" on page 16
- "Provisioning Jobs" on page 17

For more information, see "Using PlateSpin Orchestrate Jobs" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference* and the following JDL job class definitions in the same guide:

- "Job"
- "JobInfo"

Controlling Process Flow

Jobs can written to control all operations and processes of managed resources. Through jobs, the Orchestrate Server manages resources to perform work. Automated jobs (written in JDL), are broken down into joblets, which are distributed among multiple resources.

Parallel Processing

By managing many small joblets, the Orchestrate Server can enhance system performance and maximize resource use.

Managing the Cluster Life Cycle

Jobs can detect demand and monitor health of system resources, then modify clusters automatically to maximize system performance and provide failover services.

Discovery Jobs

Some jobs provide inspection of resources to more effectively management assets. These jobs enable all agents to periodically report basic resource facts and performance metrics. In essence, these metrics are stored as facts consisting of a key word and typed-value pairs like the following example:

```
resource.loadaverage=4.563, type=float
```

Jobs can poll resources and automatically trigger other jobs if resource performance values reach certain levels.

The system job scheduler is used to run resource discovery jobs to augment resource facts as demands change on resources. This can be done on a routine, scheduled basis or whenever new resources are provisioned, new software is installed, bandwidth changes occur, OS patches are deployed, or other events occur that might impact the system.

Consequently, resource facts form a capabilities database for the entire system. Jobs can be written that apply constraints to facts in policies, thus providing very granular control of all resources as required. All active resources are searchable and records are retained for all off-line resources.

The following `osInfo.job` example shows how a job sets operating system facts for specific resources:

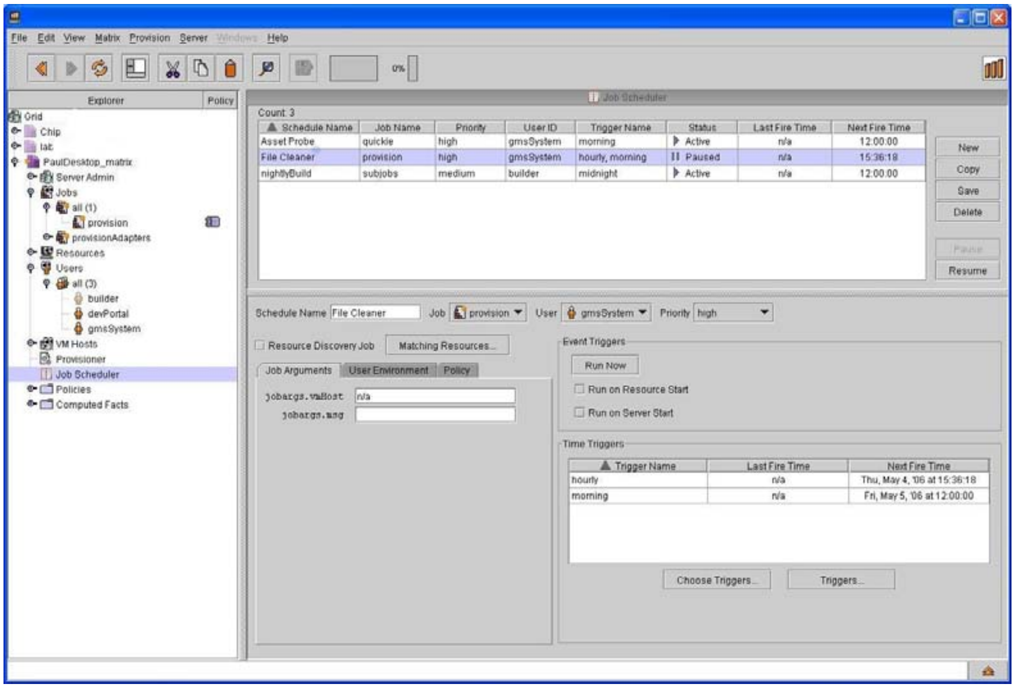
```
resource.cpu.mhz (integer) e.g., "800" (in Mhz)
    resource.cpy.vendor (string) e.g. "GenuineIntel"
    resource.cpu.model (string) e.g. "Pentium III"
    resource.cpu.family (string) e.g. "i686"
```

`osInfo.job` is packaged as a single cross-platform job and includes the Python-based JDL and a policy to set the timeout. It is run each time a new resource appears and once every 24 hours to ensure validity of the resources. For a more detailed review of this example, see "osInfo.job" in "Using PlateSpin Orchestrate Jobs" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

System Jobs

Jobs can be scheduled to to periodically trigger specific system resources based on specific time constraints or events. As shown in the following figure, PlateSpin Orchestrate provides a built-in job scheduler that enables you or system administrators to flexibly deploy and run jobs.

***Figure 1-4** The Job Scheduler*



Count: 3

Schedule Name	Job Name	Priority	User ID	Trigger Name	Status	Last Fire Time	Next Fire Time
Asset Probe	quickie	high	gmsSystem	morning	Active	n/a	12:00:00
File Cleaner	provision	high	gmsSystem	hourly, morning	Paused	n/a	15:36:18
nightlyBuild	subjobs	medium	builder	midnight	Active	n/a	12:00:00

Schedule Name: File Cleaner Job: provision User: gmsSystem Priority: high

☐ Resource Discovery Job

jobargs.vmHost: n/a

jobargs.msg:

Event Triggers

☐ Run on Resource Start

☐ Run on Server Start

Time Triggers

Trigger Name	Last Fire Time	Next Fire Time
hourly	n/a	Thu, May 4, '06 at 15:36:18
morning	n/a	Fri, May 5, '06 at 12:00:00

For more information, see "Resource Selection" in "Using PlateSpin Orchestrate Jobs" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference* and "The PlateSpin Orchestrate Job Scheduler" in the *PlateSpin Orchestrate 2.0 Development Client Reference*. See also "Job Scheduling" and "Job" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

Provisioning Jobs

Jobs also drive provisioning for virtual machines and blade servers. Provisioning adapter jobs are deployed and organized into appropriate job groups for management convenience. Provisioning adapters are deployed as part of your VMM license.

For more information, see "Virtual Machine Management" in the "PlateSpin Orchestrate 2.0 Developer Guide and Reference" and Section 1.2.1, "Resource Virtualization," on page 20.

1.1.5 Constraint-Based Job Scheduling

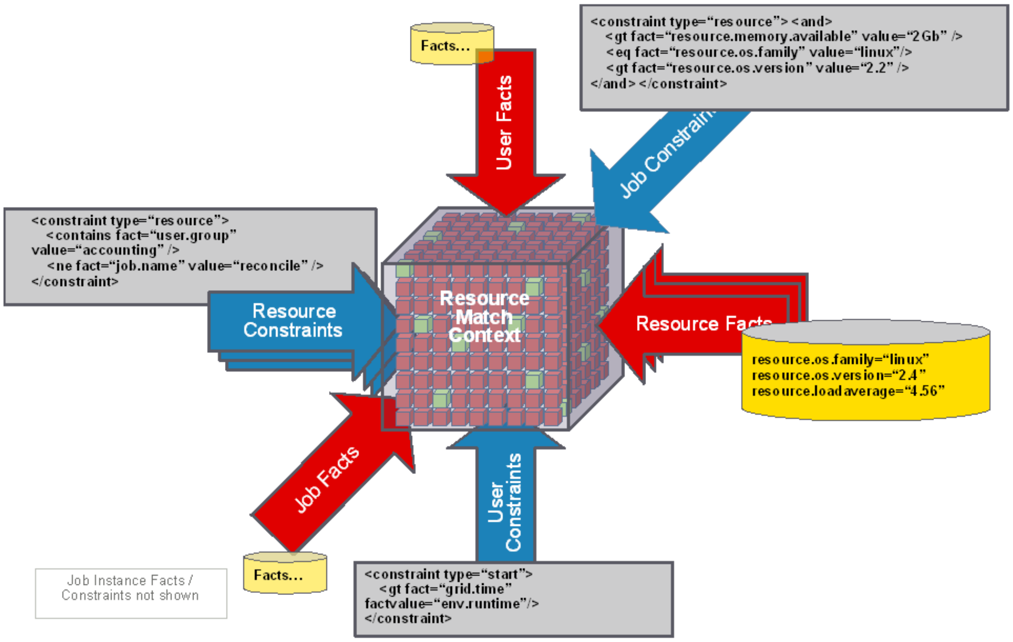
The Orchestrate Server is a "broker" that can distribute jobs to every "partner" agent on the grid. Based on assigned policies, jobs have priorities and are executed based on the following contexts:

- User Constraints
- User Facts
- Job Constraints
- Job Facts
- Job Instance
- Resource User Constraints

- Resource Facts
- Groups

Each object in a job context contains the following elements:

Figure 1-5 *Constraint-Based Resource Brokering*



```
<constraint type="resource"> <and>
   <gt fact="resource.memory.available" value="2Gb" />
   <eq fact="resource.os.family" value="linux"/>
   <gt fact="resource.os.version" value="2.2" />
</and> </constraint>
```

```
<constraint type="resource">
   <contains fact="user.group"
value="accounting" />
   <ne fact="job.name" value="reconcile" />
</constraint>
```

```
resource.os.family="linux"
resource.os.version="2.4"
resource.loadaverage="4.56"
```

```
<constraint type="start">
   <gt fact="grid.time"
factvalue="env.runtime"/>
</constraint>
```

For more information, see "Working with Facts and Constraints" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

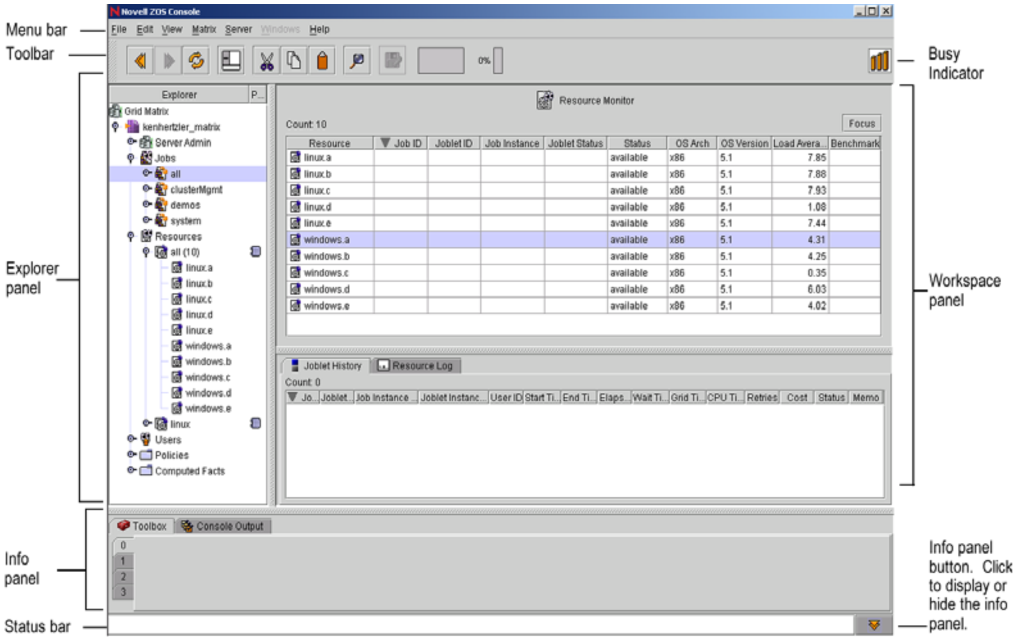
1.1.6 Understanding PlateSpin Orchestrate API Interfaces

There are three API interfaces available to the Orchestrate Server:

- **Orchestrate Server Management Interface:** The PlateSpin Orchestrate Server, written entirely in Java using the JMX (Java MBean) interface for management, leverages this API for the PlateSpin Orchestrate Development Client. The Development Client is a robust desktop GUI designed for administrators to apply, manage, and monitor usage-based policies on all infrastructure resources. The Development Client also provides at-a-glance grid health and capacity checks.

  For more information, see the *PlateSpin Orchestrate 2.0 Development Client Reference*.

Figure 1-6 *PlateSpin Orchestrate Development Client*



Count: 10

Resource	Job ID	Joblet ID	Job Instance	Joblet Status	Status	OS Arch	OS Version	Load Avera...	Benchmark
linux.a					available	x86	5.1	7.85	
linux.b					available	x86	5.1	7.88	
linux.c					available	x86	5.1	7.93	
linux.d					available	x86	5.1	1.08	
linux.e					available	x86	5.1	7.44	
windows.a					available	x86	5.1	4.31	
windows.b					available	x86	5.1	4.25	
windows.c					available	x86	5.1	0.35	
windows.d					available	x86	5.1	6.03	
windows.e					available	x86	5.1	4.02	

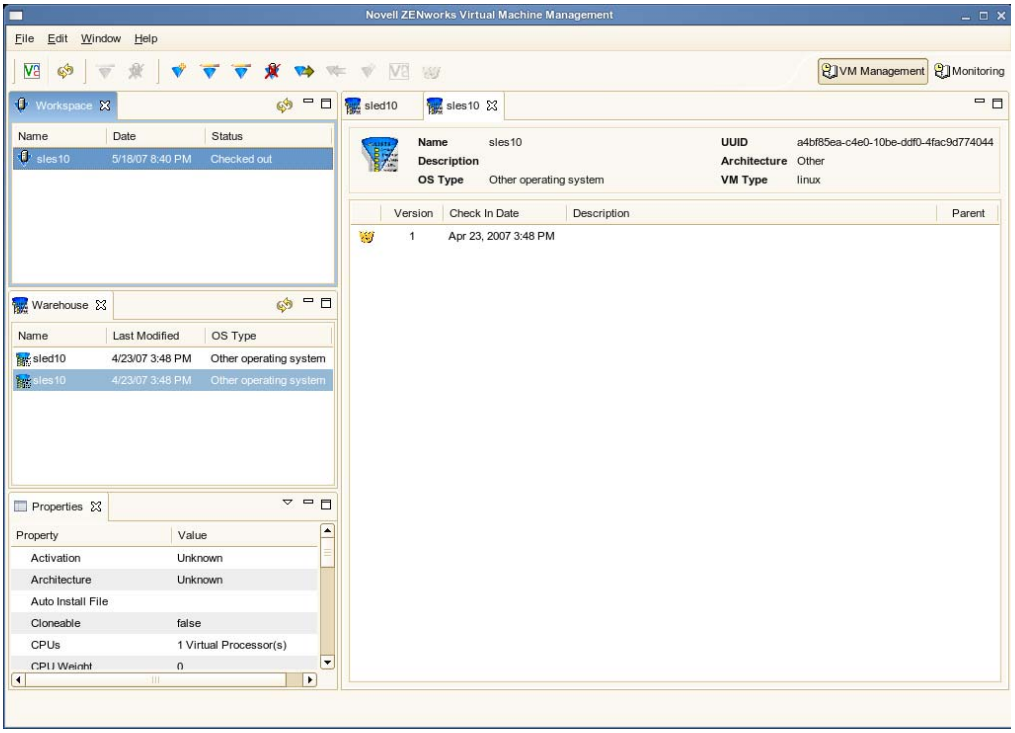
Count: 0

Jo...	Joblet...	Job Instance ...	Joblet Instanc...	User ID	Start Ti...	End Ti...	Elaps...	Wait Ti...	Grid Ti...	CPU Ti...	Retries	Cost	Status	Memo

- **Job Interface:** Includes a customizable/replaceable Web application and the zosadmin command line tool. The Web-based User Portal built with this API provides a universal job viewer from which job logs and progress can be monitored. The job interface is accessible via a Java API or CLI. A subset is also available as a Web Service. The default PlateSpin Orchestrate User Portal leverages this API. It can be customized or alternative J2EE* application can be written.
- **PlateSpin Orchestrate Monitoring System:** Monitors all aspects of the data center through an open source, Eclipse*-based interrface referred to as the PlateSpin Orchestrate VM Client.This interface operates in conjunction with the Orchestrate Server and monitors the following objects:
  - Deployed jobs that teach PlateSpin Orchestrate and provide the control logic that PlateSpin Orchestrate runs when performing its management tasks.
  - Users and Groups
  - Virtual Machines

  For more information, see the *PlateSpin Orchestrate 2.0 VM Client Guide and Reference*.

Figure 1-7 *Novell PlateSpin Orchestrate Virtual Machine Manager interface.*



Name	Date	Status
sles10	5/18/07 8:40 PM	Checked out

Name	Last Modified	OS Type
sled10	4/23/07 3:48 PM	Other operating system
sles10	4/23/07 3:48 PM	Other operating system

Property	Value
Activation	Unknown
Architecture	Unknown
Auto Install File	
Cloneable	false
CPUs	1 Virtual Processor(s)
CPU Weight	0

Name sles10
Description
OS Type Other operating system
UUID a4bf85ea-c4e0-10be-ddf0-4fac9d774044
Architecture Other
VM Type linux

Version	Check In Date	Description	Parent
1	Apr 23, 2007 3:48 PM		

1.2 Understanding PlateSpin Orchestrate Functionality

- Section 1.2.1, “Resource Virtualization,” on page 20
- Section 1.2.2, “Policy-Based Management,” on page 21
- Section 1.2.3, “Grid Object Visualization,” on page 22
- Section 1.2.4, “Understanding Job Semantics,” on page 24
- Section 1.2.5, “Distributed Messaging and Failover,” on page 25
- Section 1.2.6, “Web-Based User Interaction,” on page 26

1.2.1 Resource Virtualization

Host machines or test targets managed by the Orchestrate Server form nodes on the grid (sometimes referred to as the matrix). All resources are virtualized for access by maintaining a capabilities database containing extensive information (facts) for each managed resource.

This information is automatically polled and obtained from each resource periodically or when it first comes online. The extent of the resource information the system can gather is customizable and highly extensible, controlled by the jobs you create and deploy.

The PlateSpin Orchestrate Virtual Machine Builder is a service of VM Management that allows you to build a VM to precise specifications required for your data center. You designate the parameters required: processor, memory, hard drive space, operating system, virtualization type, whether it is based on an auto-install file, and any additional parameters. When you lauch the build job, VM Builder sends the build request to a machine that meets the hardware requirements of the defined VM and builds the VM there.

For more information, see "Creating and Setting Up Virtual Machines" in the *PlateSpin Orchestrate 2.0 VM Client Guide and Reference*.

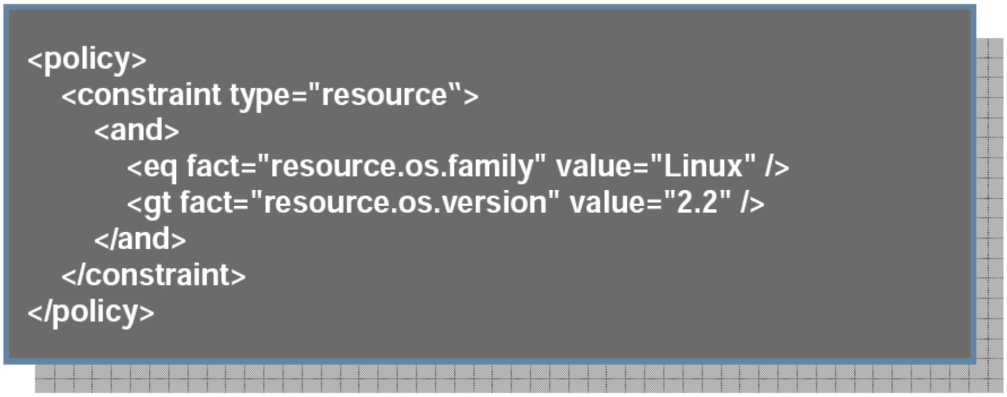
1.2.2 Policy-Based Management

Policies are aggregations of facts and constraints that are used to enforce quotas, job queuing, resource restrictions, permissions, and other user and resource functions. Policies can be set on all objects and are inherited, which facilitates implementation within related resources.

Facts, which might be static, dynamic or computed for complex logic, are used when jobs or test scenarios require resources in order to select a resource that exactly matches the requirements of the test, and to control the access and assignment of resources to particular jobs, users, projects, etc. through policies.This abstraction keeps the infrastructure fluid and allows for easy resource substitution.

Of course, direct named access is also possible. An example of a policy that constrains the selection of a resource for a particular job or test is shown in the figure below. Although resource constraints can be applied at the policy level, they can also be described by the job itself or even dynamically composed at runtime.

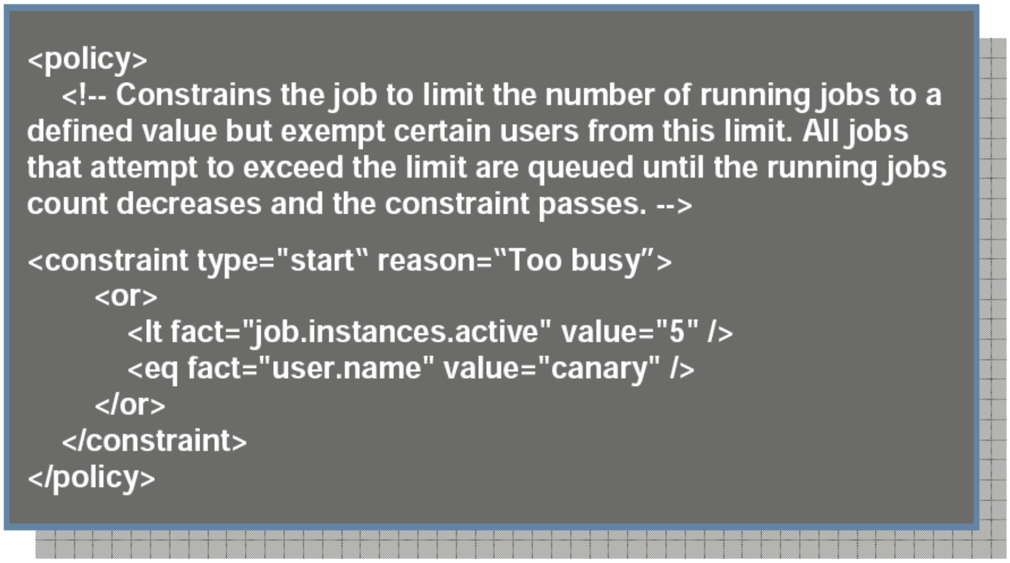
Figure 1-8 *Resource Selection Policy Example*



```
<policy>
   <constraint type="resource">
      <and>
         <eq fact="resource.os.family" value="Linux" />
         <gt fact="resource.os.version" value="2.2" />
      </and>
   </constraint>
</policy>
```

An example of a policy that constrains the start of a job or test because too many tests are already in progress is shown in the following figure:

Figure 1-9 Job/Test Start Policy Example



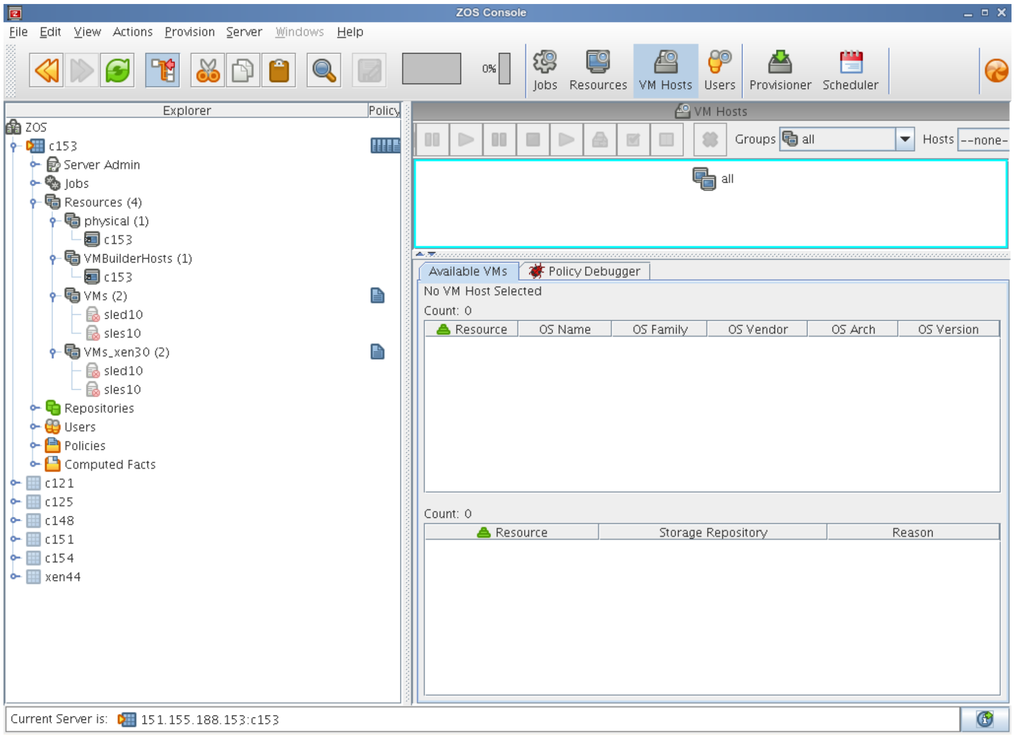
```
<policy>
   <!-- Constrains the job to limit the number of running jobs to a
defined value but exempt certain users from this limit. All jobs
that attempt to exceed the limit are queued until the running jobs
count decreases and the constraint passes. -->

<constraint type="start‟ reason=“Too busy”>
      <or>
         <lt fact="job.instances.active" value="5" />
         <eq fact="user.name" value="canary" />
      </or>
   </constraint>
</policy>
```

1.2.3 Grid Object Visualization

One of the greatest strengths of the PlateSpin Orchestrate solution is the ability to manage and visualize the entire grid. This is performed through the PlateSpin Orchestrate Development Clientand the PlateSpin Orchestrate VM Monitoring System.

The desktop Development Client is a Java application that has broad platform support and provides job, resource, and user views of activity as well as access to the historical audit database system, cost accounting, and other graphing features.

Figure 1-10 *The PlateSpin Orchestrate Development Client with Virtual Machine Management Elements*



No VM Host Selected

Count: 0

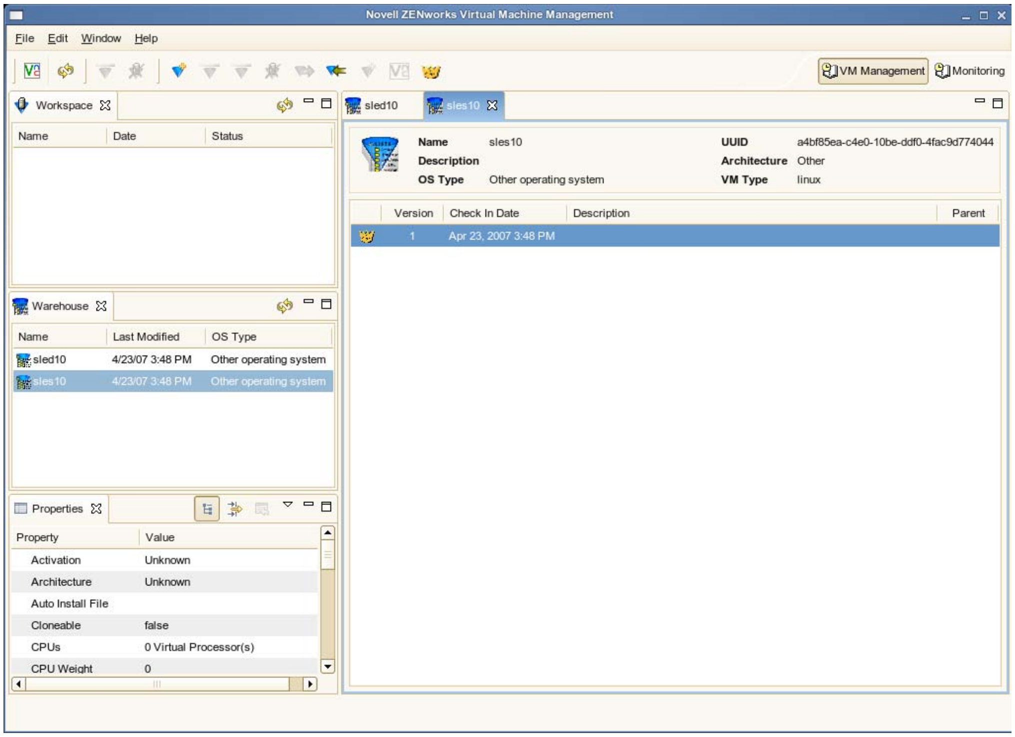
Count: 0

Current Server is: 151.155.188.153:c153

The Development Client also applies policies that govern the use of shared infrastructure or simply create logical grouping of nodes on the grid. For more information about the PlateSpin Orchestrate Development Client, see the *PlateSpin Orchestrate 2.0 Development Client Reference*.

The PlateSpine Orchestrate VM Monitoring System provides robust graphical monitoring of all managed virtual resources managed on the grid.

Figure 1-11 *The PlateSpin Orchestrate VM Monitoring System*



Name	Date	Status

Name	Last Modified	OS Type
sled10	4/23/07 3:48 PM	Other operating system
sles10	4/23/07 3:48 PM	Other operating system

Property	Value
Activation	Unknown
Architecture	Unknown
Auto Install File	
Cloneable	false
CPUs	0 Virtual Processor(s)
CPU Weight	0

Name sles10
Description
OS Type Other operating system
UUID a4bf85ea-c4e0-10be-ddf0-4fac9d774044
Architecture Other
VM Type linux

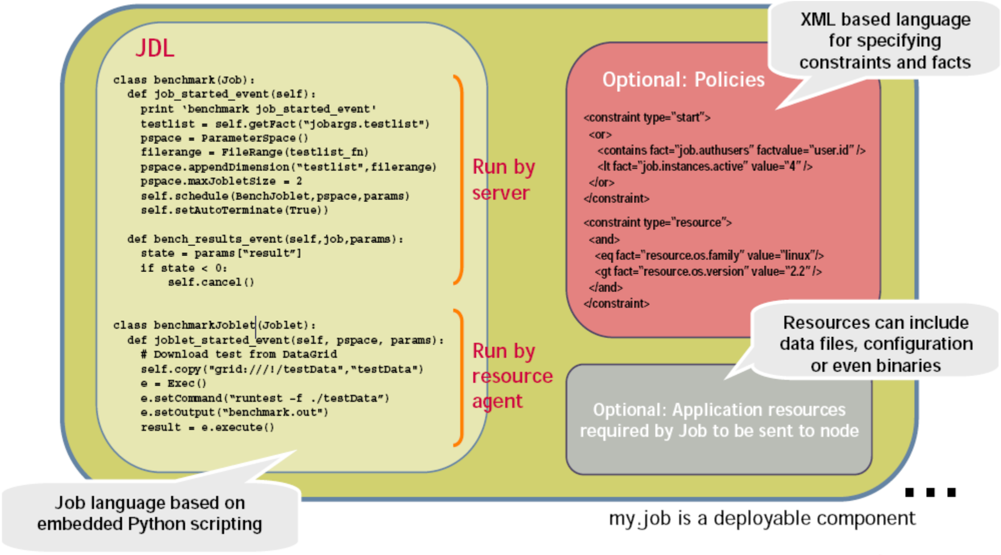
Version	Check In Date	Description	Parent
1	Apr 23, 2007 3:48 PM		

For more information, see the *PlateSpin Orchestrate 2.0 VM Client Guide and Reference*.

1.2.4 Understanding Job Semantics

As mentioned earlier, PlateSpin Orchestrate runs jobs. A job is a container that can encapsulate several components including the Python-based logic for controlling the job life cycle (such as a test) through logic that accompanies any remote activity, task-related resources such as configuration files, binaries and any policies that should be associated with the job, as illustrated below.

Figure 1-12 *Components of a Job*



```
class benchmark(Job):
  def job_started_event(self):
    print 'benchmark job_started_event'
    testlist = self.getFact("jobargs.testlist")
    pspace = ParameterSpace()
    filerange = FileRange(testlist_fn)
    pspace.appendDimension("testlist",filerange)
    pspace.maxJobletSize = 2
    self.schedule(BenchJoblet,pspace,params)
    self.setAutoTerminate(True))

  def bench_results_event(self,job,params):
    state = params["result"]
    if state < 0:
        self.cancel()


class benchmarkJoblet(Joblet):
  def joblet_started_event(self, pspace, params):
    # Download test from DataGrid
    self.copy("grid:///!/testData","testData")
    e = Exec()
    e.setCommand("runtest -f ./testData")
    e.setOutput("benchmark.out")
    result = e.execute()
```

```
<constraint type="start">
 <or>
   <contains fact="job.authusers" factvalue="user.id" />
   <lt fact="job.instances.active" value="4" />
 </or>
</constraint>

<constraint type="resource">
 <and>
  <eq fact="resource.os.family" value="linux"/>
  <gt fact="resource.os.version" value="2.2" />
 </and>
</constraint>
```

XML based language for specifying constraints and facts
Resources can include data files, configuration or even binaries
Optional: Application resources required by Job to be sent to node
Job language based on embedded Python scripting
my.job is a deployable component

Workflows

Jobs can also invoke other jobs, creating hierarchies. Because of the communication between the job client (either a user/user client application or another job) it is easy to create complex workflows composed of discrete and separately versioned components.

When a job is executed and an instance is created, the class that extends job is run on the server and as that logic requests resources, the class(es) that extend the joblet are automatically shipped to the requested resource to manage the remote task. The communication mechanism between these distributed components manifests itself as event method calls on the corresponding piece.

For more information, see "Workflow Job Example" in "Advanced Job Development Concepts", and "Job State Transition Events", or "Communicating Through Job Events" in "Job Architecture" in the *PlateSpin Orchestrate 2.0 Developer Guide and Reference*.

1.2.5 Distributed Messaging and Failover

A job has control over all aspects of its failover semantics, which can be specified separately for conditions such as the loss of a resource, failure of an individual joblet, or joblet timeout.

The failover/health check mechanisms leverage the same communications mechanism that is available to job and joblet logic. Specifically, when a job is started and resources are employed, a message interface is established among all the components as shown in Figure 1-13 on page 26.

Optionally, a communication channel can also be kept open to the initiating client. This client communication channel can be closed and reopened later based on jobid. Messages can be sent with the command

```
sendEvent(foo_event, params, ...)
```

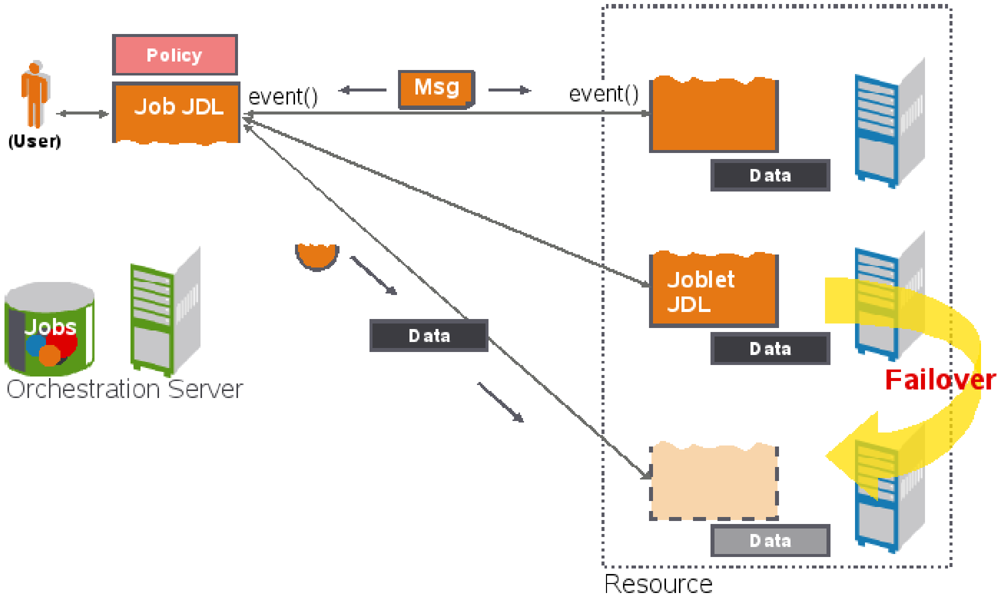
and received at the other end as a method invocation

```
def foo_event(self, params)
```

If a job allows it, a failure in any joblet causes the Orchestrate Server to automatically find an alternative resource, copy over the joblet JDL code, and reestablish the communication connection. A job also can listen for such conditions simply by defining a method for one of the internally generated events, such as def joblet_failure_event(...).

Such failover allows, for example, for a large set of regression tests to be run (perhaps in parallel) and for a resource to die in the middle of the tests without the test run being rendered invalid. The figure below shows how job logic is distributed and failover achieved:

Figure 1-13 *A Job in Action*



1.2.6 Web-Based User Interaction

PlateSpin Orchestrate ships a universal job monitoring and submission interface as a Web application that natively runs on the Orchestrate Server. This application is written to the PlateSpin Orchestrate job management API and can be customized or replaced with alternative rendering as required.The figure belows shows an example of this interface, called the *User Portal*. For more information, see the *PlateSpin Orchestrate 2.0 Job Manager Guide*.

<<add screenshot of User Portal here>>

PlateSpin Orchestrate Security

A

This section explains various security issues related to PlateSpin® Orchestrate from Novell®:

A.1 User and Administrator Password Hashing Methods

All passwords stored in PlateSpin Orchestrate are hashed using Secure Hash Alogrithm-1 (SHA-1). However, user passwords are no longer hashed when sent from the client to the server. Instead, the plain text password entered by the user is sent over an encrypted authentication connection to the server to obtain a unique per-session credential issued by the server. This allows the server to "plug in" to alternative user directories such as Active Directory or OpenLDAP. Agent credentials are still stored, singly hashed, on the disk on the agent machine. The first pass hashing prevents "user friendly" passwords entered by administrators from being compromised by storing them on the agent machines. The server's password database (for agents and for users not using an alternative user directory) stores all passwords in a double-hashed form to prevent a stolen password database from being used to obtain passwords.

WARNING: The zosadmin command line and the PlateSpin Orchestrate Development Client do not use SSL encryption, nor do they support TLS/SSL, so they should only be used over a secure network.

All agent and client connections support TLS encryption. This includes the zos command line and the PlateSpin Orchestrate Agent.

A.2 User and Agent Password Authentication

The PlateSpin Orchestrate Server stores all user and agent passwords in its data store as double-hashed strings. User clients such as the zos command send the plain text password over a TLS encrypted authentication connection to obtain a randomly generated per-session credential issued by the server. This session credential is retained by the client, either in memory or in a temporary disk file for the duration of the session.

It is not possible to obtain the user's password from the session credential, however. It should be protected to prevent unauthorized users from taking over the session. Agents send a singly hashed password as their login credential, which is in turn hashed once more on the server to authenticate new agent connections. Upon authentication, agents receive the same type of session credential as user clients.

Singly-hashed password strings are used as a special case for agents, because agents typically must store their plain text credentials to disk to allow the agents to start up on host or VM reboot. The use of a once hashed version of the password on the agent prevents administrators from compromising “user friendly” text passwords by storing them unhashed on agents. The use of single hashing on the agents and double hashing on the server database prevents stolen credential data from being used to obtain actual user or administrator-entered passwords

A.3 Password Protection

You should take measures to protect the passwords and credentials on both the PlateSpin Orchestrate Server and PlateSpin Orchestrate Agents by ensuring that only the user account of the PlateSpin Orchestrate Server (currently `root` or `Administrator`, by default) has access to the `/store` and `/tls` directories on the server, so that general users are prevented from obtaining the password. On agents, allow only the agent users (normally `root` or `Administrator`) to have access to the `agent.properties` file, which contains the agent’s authentication credential.

Currently, PlateSpin Orchestrate restricts file access on the server, but we recommend that you disallow shell accounts on server machines for general users as a precaution.

For users, none of the Novell-provided client utilities stores the user-entered password to disk in either plain text or hashed form. However, temporary once-per-session credentials are stored to the disk in the users `$HOME/.novell/zos/client` directory. Theft of this session credential could allow someone else to take over that user session, but not to steal the user’s password. Users can protect their logged-in session by making sure the permissions either on their home directory or on the `~/.novell/zos/client` directory are set to forbid both read and write access by other users.

PlateSpin Orchestrate Agents use the same authentication protocol and password hashing as users (agent passwords are stored to disk in hashed form, not plain text) with the exception that agent passwords are not salted, allowing agents to be renamed by the server. Because agent passwords are not salted, we recommend that you generate and use random non-mnemonic strings for agent passwords.

For PlateSpin Orchestrate 2.0 and later, administrators can enhance security when configuring new agents by setting the `zos.agent.password` property to the asterisk character (*). This causes the agent to automatically generate a new random credential not based on any easily guessable plain text word. When the new agent is “accepted” by the administrator, the newly generated credential is stored by the server. This is the default behavior when the PlateSpin Orchestrate Agent is first installed.

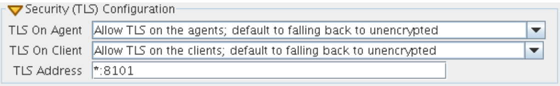
n addition, the `zos.agent.password` property can be set to a plain text password in `agent.properties`. If this is done, the agent automatically replaces the plain text password with the hashed version when it next starts. This allows administrators to more easily set up an initial password for agents.

A.4 TLS Encryption

PlateSpin Orchestrate 1.3 uses Transport Layer Security (TLS) to provide encryption for both user and agent connections. Both the PlateSpin Orchestrate Agent and the PlateSpin Orchestrate Clients use TLS to initiate their connections to the Orchestrate Server, and then the server specifies whether to “fall back” to plain text or continue the session fully encrypted.

Although you can manually configure the agent and clients to either always require TLS encryption or to fully disable TLS encryption, we recommended that you leave the agents and clients in their default configuration, and then use the PlateSpin Orchestrate Development client on the server to specify the default behavior. This is the purpose of the TLS Options section on the main server tab of the Orchestrate Development Client.

Figure A-1 *TLS Options in the PlateSpin Orchestrate Development Client*



Security (TLS) Configuration

TLS On Agent: Allow TLS on the agents; default to falling back to unencrypted

TLS On Client: Allow TLS on the clients; default to falling back to unencrypted

TLS Address: *:8101

Here, there are 4 levels that you can set separately for both agent connections and user/client connections:

- **Forbid TLS for (agents/clients):** This option is to fully disable and prohibit TLS encryption altogether. This is the least secure option and is therefore usually not the desirable choice, but it could be required in countries that restrict encryption or in low security environments where performance is more critical than security.
- **Allow TLS on the (agents/clients); default to falling back to unencrypted:** This option (the factory default for both agents and clients) is to allow TLS encryption if the agent or client explicitly requests it, but to default to falling back to plain text after authentication.

  NOTE: Authentication always occurs over SSL, regardless of settings.

- **Allow TLS on the (agents/clients); default to TLS encrypted if not configured encrypted:**

  This option is similar to the second option. Agents/clients may specify whether or not to use TLS, but if they use the default of “server specified,” the server defaults to using TLS.
- **Make TLS mandatory on the (agents/clients):** This option is the most secure, locked down option. It requires TLS at all times, and fails connections if the agent or the client tries to specify plain text.

In addition to these settings for TLS configuration, there are files that need to be protected on both the server and on the client/agent. For more information, search for the *TLS Certificate Installation On PlateSpin Orchestrate* article at the *Novell Cool Solutions Community (http://www.novell.com/communities/coolsolutions/)*.

A.5 Security for Administrative Services

The PlateSpin Orchestrate Development Client and the zosadmin command line tool are clients to the MBean and RMI servers. PlateSpin Orchestrate does not provide encryption for these administrative services, so you should be careful to use them only in a secure environment.

When the user logs in using either `zosadmin login` or the Orchestrate Development Client, the user’s password is sent to the server, and then the server issues a per-session credential to be used for further operations. The user’s cleartext password is never stored to disk; however, it is currently sent “over the wire” in plain text form. For this reason, the administrative clients should only be used in a secure, trusted environment.

The zosadmin client stores the session credential obtained from a `zosadmin login` request in a temporary file for use by subsequent operations. This credential cannot be used to obtain the user's password, but it could be used to take over the user's current session until it times out or expires. For this reason, the files in the user's `.novell/zoc/` directory should be configured to disallow access by other users.

A.6 Plain Text Visibility of Sensitive Information

The following table outlines where sensitive information might be visible as plain text:

Table A-1 *Locations Where Sensitive Information Might Be Stored As Plain Text*

Information	Storage Location	Visibility Issue
Audit Database configuration	ZOS properties store	Contains plain text information including user/password for allowing PlateSpin Orchestrate to log into the Audit Database for logging. You should use a non-privileged database account for logging.

Server Discovery and Multicasting

B

The PlateSpin* Orchestrate Server, Orchestrate Agent, and other Orchestrate tools use IP multicast messages to locate servers and to announce when servers are started or shut down. If multicasting is not supported in your existing network environment, all PlateSpin Orchestrate components allow a specific machine to be specified instead of using multicast discovery. Multicast support is not required to run the PlateSpin Orchestrate Server, the Orchestrate Agent, or any Orchestrate tools.

This section includes the following multicast information:

B.1 Multicast Troubleshooting

An exhaustive tutorial of IP multicasting and troubleshooting is beyond the scope of this document. If you are having problems with multicast discovery, ensure that your operating system is configured to provide IP multicasting support. Most modern versions of Linux* and Windows* provide this support, however it might be disabled. Multicast discovery does not work unless IP multicasting is enabled by the operating system. Routing misconfiguration on the system can also lead to problems with multicast discovery.

B.2 Multicast Routes

A common problem with multicasting, particularly on Linux, is the lack of a default route or multicast network route. Most systems are configured to have at least a “default” route, and on such systems, multicast messages use the default route like any other network traffic. Systems do not necessarily require a default route. Multicasting might not function correctly on systems that lack a default route. Attempts to send messages on such systems fail with a `Network Unreachable` message because the operating system is unable to determine the correct network interface on which to send the message.

The quick solution is to add a default route on such systems. In some environments, however, it might not make sense to add a default route. In such cases, another solution is to add a network route for the 224.0.0.0/4 block representing the multicast IP address space. On Linux, for example, issue the following command as the `root` user:

```
route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0
```

This command tells the system to send all multicast datagrams to the `eth0` network card by default. Substitute `eth0` with a different interface name if applicable.

B.3 Multi-homed Hosts

A multi-homed host is a machine with more than one network interface configured. This can be anything from a Linux system being used as a network router to a laptop computer with both an active Ethernet connection and an active wireless connection. If there are two or more network interfaces active at the same time (even if only one is actually being used) the system is "multi-homed."

Some operating systems like Linux provide only very rudimentary routing as a default part of the operating system. They rely on external routing software like "mrouted" to support full multicast routing. As a result, problems might arise in multi-homed machines because outgoing multicast messages are sent on only one interface in the absence of more sophisticated routing software. It is possible that the interface chosen by the operating system is incorrect. The Orchestrate Server and its associated tools make a best effort to ensure that discovery queries and announcements are sent on all available interfaces. It should not be necessary to run an external routing program with the current Orchestrate Server.

B.4 Multiple Subnets

By default, Orchestrate Server and its associated tools are configured to allow multicast messages to pass through up to two gateways. This allows discovery to work in multi-subnet environments, provided that the network routers on your network are properly configured to perform multicast routing. Consult the vendor's documentation for information on configuring multicast routing on your network routers.

B.5 Datagrid and Multicasting

The PlateSpin Orchestrate datagrid facility provides a multicast-based file distribution service that allows large multi-gigabyte files to be simultaneously delivered to a large number of recipient machines while using far less network bandwidth than would be used by copying the file individually to each node. This service is available only in network environments that support IP multicasting. Aside from the file multicast service, all other features of the datagrid use normal unicast network operations and do not require multicast support. The routing and troubleshooting pointers provided above for network discovery also apply to datagrid multicasting. In addition, due to the potentially large bandwidth used by file transfers, you might want to limit the set of interfaces on which files are multicasted.

B.6 Datagrid Multicast Interface Selection

On multi-homed servers, the datagrid multicast service sends outbound control and data packets on all available interfaces on the system. This allows datagrid multicasting to work "out of the box" with multi-homed servers. This behavior might not be optimal if you require multicasting of files only to a subset of the available interfaces. You can instruct the datagrid to multicast only to the desired interfaces by selecting the correct interfaces from the Orchestrate Development Client on the Info/Configuration tab under *Data Grid Configuration*. Restricting the set of datagrid multicast interfaces prevents large amounts of file data from being sent to uninterested subnets.