IGI PUBLISHING
701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.igi-pub.com

ITJ3833

This paper appears in the publication, International Journal of Web Services Research Volume 4, Issue 3
edited by **Liang-Jie Zhang © 2007, IGI Global**

Dealing with Scale and Adaptation of Global Web Services Management

William Vambenepe, HP, USA
Carol Thompson, HP, USA
Vanish Talwar, HP, USA
Sandro Rafaeli, HP, USA
Bryan Murray, HP, USA
Dejan Milojicic, HP, USA
Subu Iyer, HP, USA
Keith I. Farkas, VMware, USA
Martin Arlitt, HP, USA

ABSTRACT

Service Oriented Architectures (SOA) are becoming the prevalent approach for realizing modern services and systems. SOA offers superior support for autonomy (decoupling) and heterogeneity compared to previous generation middleware systems, resulting in more scalable and adaptive solutions. However, SOA have not adequately addressed management, while traditional management solutions do not sufficiently scale to address the needs of (global) Web services. We propose scalable management based on models and industry standards. We discuss a use case for global service management and present its design, implementation, and preliminary evaluation. We retain all the benefits of SOA while also enabling global scale manageability. Our approach provides manageability that is comprehensible for administrators yet automated enough for integration into autonomous systems.

Keywords: adaptation; PlanetLab; scale; standards; Web services management

INTRODUCTION

The increasing scale and complexity of systems and services makes them increasingly difficult and expensive to administer. Service Oriented Architectures (SOA) (Huhns & Singh, 2005) contributed to overcome these problems, but they do not sufficiently address the management of services.

Updating a moderately sized data center may require changes to software on thousands of machines. In the case of global services in a large enterprise, a software update may require touching hundreds of data centers. In addition, the complexity of these services increases as there may be interdependencies among the services. For example, a Web-based e-commerce application may consist of a virtual store, catalog, customer relationship, and billing services, among many others. At the infrastructure level, this application is usually mapped on a three-tier system architecture, comprising the database, application, and Web server tiers. The application tier further consists of the application server, the application in question, and other services on which the application depends. Large scale data centers in financial, public and private sector, etc. can be significantly larger in size with significantly more complex services.

In addition, traditional enterprise data centers are being complemented with so called closet computers emerging from remote and home offices. New computing models, such as Utility Computing (Wilkes, Mogul, & Suermondt, 2004) (Kandlur & Killela, 2004), Grid Computing (Foster, Kesselman, Nick, & Tuecke, 2002), and PlanetLab (Peterson, Anderson, Culler, & Roscoe, 2002) grow even more significantly in scale.

Availability needs change as companies move from expensive, private networks with well-defined management policies to the Internet and poorly defined policies and best practices. Such shifts require adaptation to unexpected loads, rebooting and upgrading of machines, networks, and services. As the systems continue to grow in size and global deployment, the traditional management approaches become less effective. To address these new requirements, we propose a new way of scalable management, based on the use of models and standards- based interfaces. The work presented in this article is related to our work on approaches to service deployment and on scalable communication described elsewhere (Adams et al., 2005; Talwar et al., 2005).

The rest of the article is organized in the following manner. First, we overview related standards in the management area. We then present a use case scenario. Subsequently, we describe our solution and discuss model federation. We then evaluate our solution followed by lessons learned and related work. Finally, we summarize our contributions and discuss future work.

INDUSTRY STANDARDS BACKGROUND

Our work relies on the use of industry standards in order to ensure that there is interoperability between long-lived global services as well as infrastructures they execute on. In this section we provide a summary of standards in the area of models, management, deployment workflows, and security.

Web based enterprise management (**WBEM**) is a set of management standards for distributed computing environments, developed by the Distributed Management Task Force, Inc. (DMTF; www.dmtf.org/standards/wbem). WBEM has been designed to simplify system management across multiple computing environments. The core set of WBEM standards includes the common information model (**CIM**) standard, a data model for representing common management information for systems, networks, applications, services, and the dependences between these components (www.dmtf.org/standards/cim). CIM specifies a schema, which provides the definitions of the model, and a metaschema, which facilitates integrating CIM with other models.

The Web services distributed management (**WSDM**) technical committee in OASIS produced the Management Using Web Services (MUWS) specification to describe a standard way to advertise, expose and access manageability capabilities through Web services (www.oasis-open.org/committees/wsdm/charter.php). The specification defines notions such as manageable resources, manageability endpoints, and manageability capabilities. It provides a common way to handle manageability endpoints and assess their identity. Management models such as CIM can make use of WSDM MUWS to make their semantics available through the standard mechanism for exposing management information through Web services.

The GGF's Configuration Description, Deployment, and Lifecycle Management Working Group (**CDDLM**-WG), pursues Web service deployment in the Grid space (https://forge.gridforum.org/ projects/cddlm-wg) . The CDDLM deployment is an extension of the

OASIS WSDM. CDDLM defines a language for specifying deployment requests, the component model that enables services to become deployable, and a set of Web services interfaces (in WSDL) for invoking deployment. CDDLM reference implementations are in progress and we plan to use then once they become more reliable.

The business process execution language (**BPEL**) is a standard published by OASIS (www.oasis-open.org/committees/wsbpel/charter.php). BPEL for Web services is an XML-based language designed to enable task-sharing for distributed computing. BPEL orchestrates Web Services by specifying the order in which it is meaningful to invoke a collection of services. A Business Process in BPEL is composed of several Web Service invocations, Receptions, and Decision Points with simple conditional logic and parallel flows or sequences.

OASIS Web services security (**WSS**) TC produced the Web services security (**WS-Security**) set of specification to enable standard use of existing security technologies such as X.509 certificates, Kerberos tickets and SAML Assertions to enhance SOAP messages (www.oasis-open.org/committees/ tc_home.php?wg_abbrev=wss). **WS-Trust** (Web services trust language) extends WS-Security for issuing security tokens and credentials in different trust domains.

USE CASE: GLOBAL SERVICES MANAGEMENT

In this section, we consider a scenario involving the deployment of a global scale, three-tier e-commerce application. The scenario consists of deploying the application onto a large number of nodes. Some nodes support the database, while others support the Web and e-commerce application. The Web application is configured to connect to the correct database node. It is possible, in case of failures, to reconfigure the Web application to migrate to a different database server. A few nodes have other services running, and these services use the default ports of the Web server and database server, which means that the deployed application has to be configured to run on a different port. The database and Web applications are customized for geographic location. For example, the application running on a node in Brazil is presented in Portuguese and should offer products that are relevant to Brazilian people. Configuring the correct language requires detecting the language to use and then activating the proper Web application files as well as filling the database with the correct product catalogue.

Furthermore, the nodes assigned for a globally distributed application would typically span several distinct trust realms. This means that the nodes on which the application is being deployed may not be able to directly identify, authenticate or authorize the deployment engines because of the differing security mechanism among them. In this scenario, there is also the possibility of failures. These faults can happen to one or more Web application, database or Web servers. There is some way to monitor the deployed applications in order to detect these failures and then take some action to solve the problem. This scenario requires the following:

1. Web-services-based scalable deployment: for decoupled and scalable communication.
2. Model-based configuration and adaptation: for machine-readable system configuration.
3. Event-based notification: for scalable failure notification (pursued elsewhere by Brett et al., 2004).
4. Security framework for providing authentication, authorization, integrity checking and confidentiality. The security framework is required to provide cross-domain authentication and authorization.

In our work, the core service model is the same in all deployments, and the localization is modeled as an extension to the core model. This allows us to configure, deploy and manage the services in a coherent manner, maintaining a consistent view of the deployments, regardless of customization. Furthermore, using the standard manageability interfaces enables the components to configure each other on an as needed basis.

OUR SOLUTION

In this section, we present our solution to address the problems identified above. The system model we consider consists of a set of globally distributed nodes such as those on PlanetLab[1]. The nodes are subject to changes such as failures, upgrades throughout their lifecycle and host services catering to different geographies. Such a computing environment has characteristics of *scale*, *virtualization*, and *dynamism*.

The solution principal entities consist of deployment, health monitoring, and adaptation services (see Figure 1). In addition, we design a security framework for these services. Our overall approach is to design these services using Web services and models. The solution is based on using the industry standards introduced earlier in the article and subject them to very large scale and dynamism.

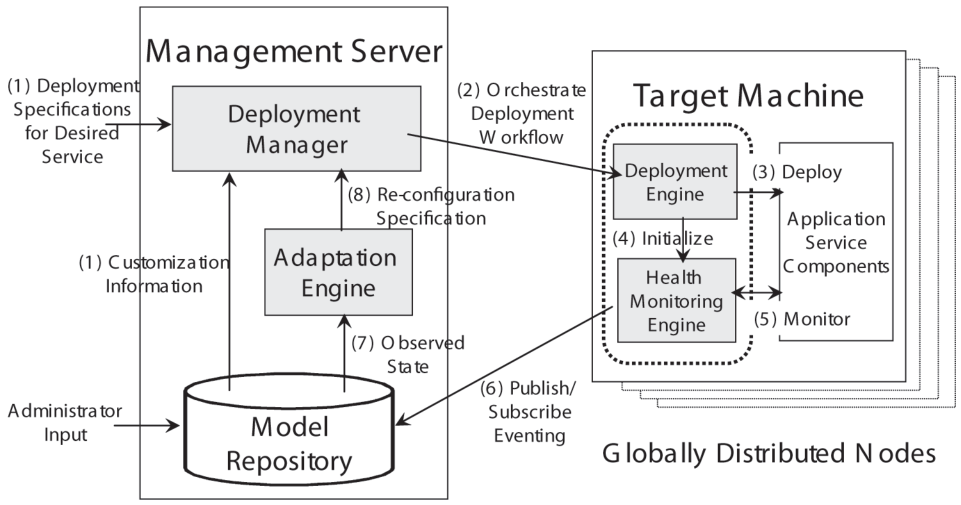
The overall execution flow in the system goes through four main phases. In the first phase, users or customers request for their services to be globally installed and instantiated. During this phase, the deployment service uses a well-defined description of the desired service specified in formal languages and orchestrates a workflow to install, configure, and activate the service. Once the service is activated, in the subsequent phase, the health monitoring service logs the service activity into event structures. This log information is then provided to a publish-subscribe eventing system such as that described in Brett et al. (2004). Next, the semantic relationships among the various events are formally represented in well-defined model structures. In the final phase, an adaptation service acts on the information in the model, and in case of policy violations, the adaptation service executes re-configuration decisions. This may require reinvoking the deployment service, if needed.

In the following subsections, we describe each of the solution entities in detail. We specifically consider instances of JPetStore as the service to be deployed globally and use it as the running example in the subsequent subsections.

Deployment Service

Our deployment service is responsible for performing the installation, configuration, activation, deactivation, and deinstallation of global

Figure 1. Solution components


Management Server
(1) Deployment Specifications for Desired Service
Deployment Manager
(2) Orchestrate Deployment Workflow
(8) Re-configuration Specification
(1) Customization Information
Adaptation Engine
(7) Observed State
Administrator Input
Model Repository
(6) Publish/ Subscribe Eventing
Target Machine
Deployment Engine
(3) Deploy
Application Service Components
(4) Initialize
Health Monitoring Engine
(5) Monitor
Globally Distributed Nodes

application services. It is primarily comprised of an infrastructure component consisting of Web services-based deployment and workflow engines; a service description component consisting of language parsers and interpreters; and an eventing component consisting of event triggers and event visualization tools.

An instance of a JPetStore testbed consists of a Tomcat server, a MySQL server, and JPetStore application files. A typical deployment process involves the download of each of these packages, the installation to appropriate folders, their configuration, and then subsequent activation. We wrote generic Java components that capture the logic for performing these actions. In order to customize the JPetStore instance based on geography, we capture the attributes for each geography in CIM models. At the time of deployment, this information is obtained from the CIM repository and mapped into the deployment configuration input file.

The Java component is designed so that it can read many of the parameters specific to an application through a configuration file. The generic Java components we wrote include GenericRPMInstaller, GenericTarInstaller, GenericActivator, GenericRSyncDownloader, and GenericFailureDetector. These components are then distributed as a library along with the deployment engine infrastructure package (see Figure 2 for the example code snippet).

The web services based deployment engine exists on all of the deployment target nodes. It receives and processes the deployment requests given to a deployment target node. Based on a deployment request, it locates the appropriate Java component responsible for a request, and then invokes the appropriate methods on that component.

At the time of deployment, we describe the specific configuration information needed during the JPetStore deployment in a well-defined deployment language. These parameters are, for example, the name of the deployment server, the package names, the destination directories, the download byte size, and so forth. The language parsers and interpreters execute at the deployment target nodes. They are invoked during the execution of the appropriate Java components at the target node.

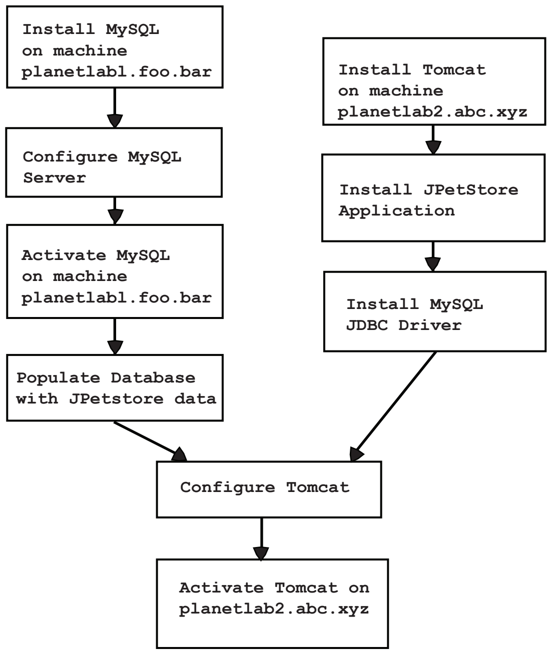
We also describe the deployment dependencies that exist among the various components of the JPetStore package as a workflow. Figure 3 shows the conceptual workflow needed for an instance of a JPetStore.

This is formally represented in a workflow language, wherein we describe the destination host, the functionality to be performed, and the

Figure 2. Snippet of deployment component

```
public class GenericRPMInstaller
{
   public boolean install(String parameters) { ....
        // download the packages
     RsyncDownloader downloader = new
     RsyncDownloader(downloadFromDir,downloadToLocation,
     new Integer(downloadBlockSize).intValue());
     downloader.download();
        // install the package
     String installCmd = rpmCmd+downloadToLocation+"/"+rpm;
     File file = new File(downloadToLocation); .....
     p = Runtime.getRuntime ().exec (installCmd,null,file); .....
   }
}
```

Figure 3. Workflow for the deployment of JPetStore


Install MySQL on machine planetlabl.foo.bar
Configure MySQL Server
Activate MySQL on machine planetlabl.foo.bar
Populate Database with JPetstore data
Install Tomcat on machine planetlab2.abc.xyz
Install JPetStore Application
Install MySQL JDBC Driver
Configure Tomcat
Activate Tomcat on planetlab2.abc.xyz

configuration language specification for the deployment step. In this workflow, we map the dependency requirements that the application service provider has specified to the actual instances of the packages and services within the system. (See Figure 4 for the example code.)

The BPEL workflow specifies a composition of tasks to be performed by the management components and it is provided to the BPEL workflow engine. The workflow engine executes at the deployment server node. It parses and processes the deployment workflow descriptions. It then invokes the deployment engines on the target nodes using SOAP. The deployment engine when thus invoked processes the deployment requests as described earlier.

Various event triggers are started during the deployment process. The event triggers are written to send notifications about START, FAILURE, and HEARTBEAT for the deployed process. These events are then visualized through visualization tools.

Health Monitoring Service

The Health Monitoring Service is responsible for monitoring the execution of application processes started on the target machine. The deployment engine tells the health monitoring service the name of process to be monitored and whatever happens to that process is reported to the adaptation service (see Figure 5).

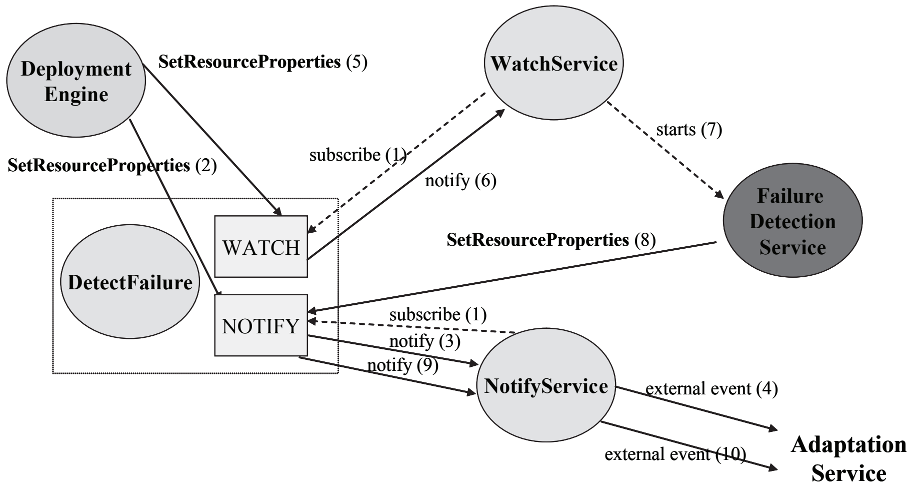
The health monitoring service is formed by three WSDM-compliant Web services. The DetectFailure Web service is just a place holder for resource properties, namely WATCH and NOTIFY. From time to time, these resource properties are updated, and DetectFailure sends notification events to the subscribers of those resource properties.

The WatchService Web service subscribes to the WATCH resource property of DetectFailure. When a notification is received, the WatchService starts a failure detection service for monitoring an application process. The NotifyService Web service subscribes to the

Figure 4. Snippet of deployment workflow specification

```
<sequence name=""main"">
<receive name=""receiveInput"" partnerLink=""client"" portType=""tns:
PLDBInstallation-Sequence"" operation=""process"" variable=""input""
createInstance=""yes""/>
.....
<invoke name=""invoke-1"" partnerLink=""deploymentengine-node-24""
operation=""invokeEngine"" portType=""nsx24:DeploymentEngine""
inputVariable=""net-xmpp_input""/>
.....
<invoke name=""invoke-2"" partnerLink=""deploymentengine-node-15""
portType=""nsx15:DeploymentEngine"" operation=""invokeEngine""
inputVariable=""net-psepr_input""/>
......
</sequence>
```

Figure 5. Health Monitoring Service



NOTIFY resource property of DetectFailure. Whatever is written to NOTIFY is then provided to NotifyService that on its turn translates the WSDM event into an external event and it is sent to adaptation service.

The Health Monitoring Service is triggered by the deployment engine (see Figure 5). Once the deployment engine has started the deployment of an application, it calls the SetResourceProperty operation (WS-ResourceProperties) on DetectFailure and sets a new value to the NOTIFY resource property.

At this moment, NOTIFY is set to a STARTUP event. The NotifyService is then notified of this event and translates it from WSDM to an external event and sends it to adaptation service. Once the deployment engine has finished deploying (started) that application, it calls the SetResourceProperty operation on DetectFailure and sets a new value to the

WATCH resource property. The WatchService is then notified of the new WATCH value and based on its content the WatchService starts a failure detection service.

On its turn, the failure detection service keeps watching the application process and generates events of the current state of that process. It calls the SetResourceProperty operation on DetectFailure and sets a new value to the NOTIFY resource property. This event is received by NotifyService and passed on to the adaptation service. There are two types of events generated by failure detection service. The first one is HEARTBEAT, which tells adaptation service that the designated application is up and running. The second event is FAILURE. This event tells the adaptation service that the process is no longer running on the target machine. After generating a FAILURE event, the failure detection service stops running.

Model-Based Adaptation Service

The implementation of the adaptation service is comprised of CIM repositories; a discovery and eventing mechanism that populates and updates the models throughout the service lifecycle; and scalable decision making services that act upon the information in the models for adaptation.

The motivation for using models is the need to capture in a structured manner the application details, the dependencies among various application components, and their relationship with the underlying hardware. For example, in a standard three-tier application, several application servers could talk to one database server. So, if the database goes down, all of the application servers connecting to this database server would also fail. We look at CIM models as a way of capturing the complex relationships between different application components.

We are using the WBEM implementation for CIM repositories. We create a model of JPetStore instances. Several instances of the JPetstore testbed exist and their attributes are each customized based on geography and internationalization. An eventing mechanism is used to receive change events from the Health Monitoring Service. This communication between the Adaptation Service and the Health Monitoring Service happens through an external publish/subscribe event system. What happens is that instead of subscribing to DetectFailure's NOTIFY resource property, the adaptation service subscribes to a single given topic on this event system. Whatever information is written to NOTIFY is translated from a WSDM event to this event system format. The design option of using an external publish/subscribe system instead of directly using WSDM's WS-Notification mechanism is driven by the scalability required by highly distributed systems.

Using WS-Notification, the adaptation service would have to subscribe to all NOTIFY resource properties on every target machine being deployed. This clearly does not scale to a large number of target machines (or nodes). However, by allowing the adaptation service subscribe to only a single topic, the burden of managing all events generated is passed to the publish/subscribe system infrastructure being used. It is assumed that such system can handle the expected number of events generated by the health monitoring service. However, using external publish/subscribe system instead of using WS-Notification would result in network and security issues. We leverage the work on eventing systems being done by Brett et al. (2004) to address these issues.

On receiving these change events, the model is updated to reflect the changes. The information is then acted upon by decision making engines. In our implementation, we perform a redeployment in case of failures. Such a redeployment action takes into consideration the dependencies that exist among various application components. In many cases, the decision making engine needs knowledge about the current state across multiple distributed nodes. We also leverage the geography information captured in the CIM models to customize the redeployment based on the location of the targeted Web service.

The whole process is prone to failures during deployment time, which means that our adaptation service could never receive any FAILURE events because the Health Monitoring Service had not been launched for the given deploying component. For these cases, we start a timeout for every node being deployed. When the timeout expires and no FAILURE or HEARTBEAT events have been received, the adaptation service assumes the node has failed completely and it starts a process of redeploying the component on another node.

In our prototype, we show the adaptation service reacting to failures of MySQL servers. The failure events are propagated through the eventing infrastructure, in response the adaptation service triggers a redeployment action, and eventually the MySQL servers are restarted.

Security Framework

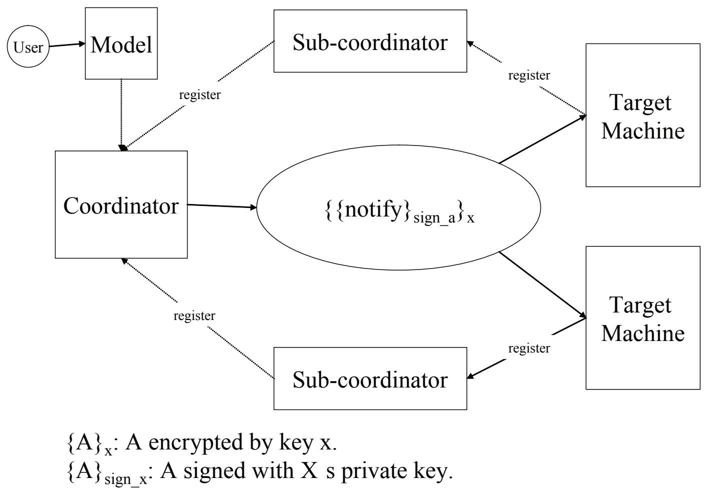
Based on the requirements defined previously, we first identify the threat model we are considering for our system:

- **Data integrity:** The data sent from deployment coordinator to deployment engines (or *target machines*) can be easily modified. Furthermore, defects sent by target machines to the deployment coordinator can be modified and false problems can be injected.
- **Impersonation:** Given the automation level we expect to achieve with our system, where adaptation engines are monitoring and dispatching adaptation scripts, the system is vulnerable to identity misuse. An intruder with malicious intentions can pose as a valid adapter and trigger hindering actions on any target machine.
- **Unauthorized access:** Certain actions require different levels of authorization.
- **Collection of sensitive data:** Nodes placed in the route between deployment entities have access to all data trafficking. The data is highly sensitive to external analysis. Computer hackers can analyze monitoring data and use this information to break into systems.

The basic security functions required to protect against the threats raised above are integrity checking, authentication, authorization and confidentiality:

- **Integrity checking:** Data integrity must be guaranteed. Any accidental corruption or intentional manipulation of the data must be detected to avoid exposure to false alarms.

Figure 6. Security architecture



- **Authentication:** The identity of any entity performing any type of interaction in the system must be properly authenticated. External entities must be prevented from interacting with valid entities. Valid entities must be treated as such. A valid entity must not be barred when trying to perform a valid operation.
- **Authorization:** Valid members must carry security tokens that state unambiguously the operations they are authorized to perform.
- **Confidentiality:** Only properly authorized entities must have access to data flowing between two entities.

Our security model has a coordinator and several target machines. The target machines register with the coordinator and receive a key also known as group key (Rafaeli & Hutchison, 2003). The group key is used to encrypt all communication among target machines and coordinator (see Figure 6). We use a public key infrastructure to enable authentication. Every domain has a root certification authority that issues certificates for all entities in the domain. We refer to such domain as a *trust domain.*

The desired security properties to be used for each WS call (operation) are defined in a configuration file. Figure 7 shows an example of this file. The Register operation is to be signed and encrypted using asymmetric key encryption (public key of destination), and Notify is to be signed and encrypted using symmetric key encryption (group key). These security policies are applied on the SOAP messages using WS-Security.

Our framework provides role-based authorization. The certificate issued to a given entity in a domain carries a CertificatePolicies extension (Housley, Ford, Polk, & Soho, 1999) that designates the role of the certificate's subject for that domain. The roles are hierarchically organized in levels, where each level

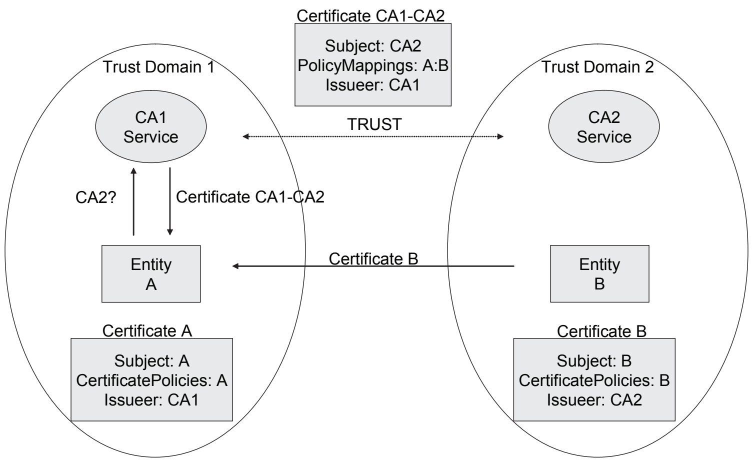
Figure 7. An example security policy file

```
<security-policy xmlns=”“http://wss.dsmt.org/xml/ns/wss/config””>
  <port ns=”“http://glue.dsmt.org/monitor””>
    <operation name=”“Register””>
      <policy>
        <sign/>
        <asymmetric-encryption/>
      </policy>
    </operation>
    <operation name=”“Notify””>
      <policy>
        <sign/>
        <symmetric-encryption/>
      </policy>
    </operation>
  </port>
  <port ns=”“http://glue.dsmt.org/federate””>
    <operation name=”“Validate””>
      <policy>
        <sign/>
      </policy>
    </operation>
  </port>
</security-policy>
```

Figure 8. An example authorization file

```
<authorization-policy xmlns=
""http://glue.dsmt.org/xml/ns/authorization"">
  <port ns=""http://glue.dsmt.org/monitor"">
    <operation name=""Register""            level=""0""/>
    <operation name=""Notify""              level=""1""/>
  </port>
  <port ns=""http://glue.dsmt.org/federate"">
    <operation name=""ValidateResponse"" level=""2""/>
  </port>
</authorization-policy>
```

Figure 9. Cross-certification



has its own set of privileges, and higher levels encompass the privileges of the lower levels. We currently have three roles: *target*, *server* and *CA* roles (respectively the lowest to the highest). Target machines are assigned to role **target**. The coordinator and sub coordinators are assigned to role **server** and the certification authority service is assigned to role **CA**.

Each operation in our system (registration, notification and federation is configured with a given level (role) and only entities assigned to the given (or higher) level are authorized to perform that operation. See Figure 8 for an example.

In order to improve the scalability of our system we use a hierarchy of subcoordinators. Target machines register with subcoordinators and not directly with the coordinator. When the group key is updated, the coordinator sends it to subcoordinators that forward it to target machines. Note that there can be a chain of subcoordinators between a target machine and the coordinator.

Federation

As we have seen previously, target machines might be placed in a trust domain different from the coordinator's trust domain. Figure 9 illustrates an example: Entity A is in trust domain 1 and entity B is in trust domain 2. Since each entity has its certificate issued by its own trust domain's certification authority, they cannot authenticate each other directly.

In our framework, we assume the existence of a trust relationship between the certification authorities of trust domains 1 and 2. This relationship is expressed in the form of cross-certification. CA1 issues a certificate for CA2 and CA2 issues a certificate for CA1. The certificates have a PolicyMappings extension [2] that maps the policy models between the two domains.

When entity A receives Certificate B, issued by CA2, which at first is not recognized by entity A, it asks CA1 to identify CA2. CA1 has issued a cross-certificate to CA2 (Certificate CA1-CA2), and then this certificate is returned to entity A. Entity A can now authenticate Certificate B and then map between Policy A and Policy B and verify if entity B has the correct privileges to execute the operation being requested.

MODEL FEDERATION

In this section, we address the scenarios of federation wherein multiple distributed model repositories exist and aggregation of data and actions from these different repositories is needed. In many cases this is done manually, through the use of complementary IT management tools, or through ad hoc integration which results in high costs. We present model federation to address these issues for large scale systems. A consistent framework for model federation is the equivalent of switching from paper maps to electronic maps. It provides a searchable, metadata-rich environment in which information can be accessed based on the boundaries of the IT system of interest to the invoker, not the layout and distribution of relevant information.

Federated repositories go beyond exposing manageability of individual resources and instead provide access to an entire system of related resources. Resources can be grouped in a system based on commonality of location, ownership, purpose or any other reason. The model access framework ensures that this grouping can be realized independently of location and implementation of manageability for the participating resources, by using Web services standards for integration. The key concepts are:

- *Model element:* an XML fragment that represents a characteristic of a *resource*. It corresponds to a CIM property, a WSDM MUWS property or an element in a WS-Management state document.
- *Resource*: A real-life (physical or logical) entity. It corresponds to a CIM instance, a WSDM MUWS manageable resource or a WS-Management resource instance.
- *System model*: An XML description of a specific system, through description of its composing *resources* and their relationships. In current standards, a system model corresponds to an XML serialization of a portion of the content of a CIMOM, a WS-ServiceGroup containing MUWS resources or a WS-Management catalog. In the general case, a system is composed of several resources. The case where a system corresponds to just one resource is logically a special case, albeit an arguably common one.

Model-driven management requires access to models. The Web services framework for accessing system models assumes that the models are represented in XML: a system model is an XML document.

In general, the Web services framework described here does not specify how the document that represents the system is created, structured or populated. It only requires that this document be represented as an XML document. Modeling standards and techniques describe how the XML document is created. For example, the CIM model describes the semantic of the model elements if the resources are described using CIM. It also describes how to represent relationship among resources (through CIM associations). A CIM to XML mapping describes how to turn this CIM description of the system into an XML document.

In the case where there is no existing resource model for the resource or the modeling framework used for the resource does not provide all the directions to create the XML representation of the system, some specifications provide generic (non resource specific) elements.

WSDM MUWS Part 2 and WS-ResourceLifetime define a set of standard model elements, such as elements to represent relationships among resources, a caption, the version, a human-readable description of the resource, the operational status of the resource, etc. Similarly, WS-ManagementCatalog defines such model elements as a display name, the name of the vendor for the resource. and so forth.

In some cases, there is a resource model for the resources but the resource model only provides ways to represent individual resources, not to generate an XML document that represents the entire system. For example, the CIM model provides classes for many types of resources but assumes that the system model will be accessed object by object, using the interfaces defined by the WBEM framework. It does not, at least at this point, provide a way to generate one XML document that represents the content of a CIMOM (or a portion of it larger than just one instance).

For such cases where the resource model does not provide a way to aggregate resources to provide a representation of the system, WS-ServiceGroup provides one way to create that logical XML document. In this case the system model is the resource properties document of a service group that contains a set of resources. The relationships among these resources are represented by model elements in the representation of the resources. For example, through model elements defined by the resource model (e.g., CIM associations) or through MUWS relationships elements.

While, as illustrated above, some of the standards and specifications intended for Web services -based management can offer help in creating the system model document, the major value provided by the Web services stack is in accessing the resource model. This is done through a set of specifications defining aspects of the SOAP messages used to interact with the system model. Looking at it through this perspective reveals how similar WS-ResourceProperties and WS-Transfer/WS-Enumeration are. Both specifications define SOAP messages to retrieve the entire system model. In addition, they provide ways to retrieve portions of the document through an XPath-based mechanism (for WS- Transfer this is done through an extension currently defined in WS-Management). WS-ResourceProperties also provides special operations for the common special case of retrieving children of the top level element of the XML representation of the system.

While the nonnormative text of these specifications seems to imply that the messages are used to retrieve the description of a unique "resource", the term "resource" in that context can be applied to anything that is described by an XML document. A system model can therefore be such a "resource", it is not limited to resources as real-life entities.

One of the key principles of the Web services architecture is to minimize the coupling between participants. Tight coupling creates possible breakage points. In addition to abstracting out programming languages and operating systems, contracts of services should also be designed to minimize assumptions between participants. One element of this is the use of WS-addressing endpoint references (EPRs). An EPR contains all the information needed to address a given endpoint. The way to process an EPR to extract and make use of this information is described by the WS-Addressing specification; it does not require any knowledge of the specificities of the endpoint. Thus a service consumer can be written to make no assumption about the way to address a resource as the service consumer only assumes that it will be handled an EPR that contains the information. Should the way the service is addressed change, the service consumer will not need to change its implementation, it will just need to be provided with an up to date EPR for the service. Thus another potential breakage point is removed: the service consumer only needs to understand the semantics of the messages exchanged, not the semantics of the headers used for addressing. On the other hand, it requires EPRs to be discovered from an authoritative source.

In the case of management interactions for retrieving elements of a system model, understanding the semantics of the messages exchanged translates to understanding the model of the system. In addition, in many cases the addressing of the resources is based on elements of the model.

Figure 10 shows an example of a message that could be used to retrieve the state of an instance of the JPetStore application (specific syntax of the body would vary depending on whether WS-Transfer or WS-ResourceProperties is used). A likely reply is presented in Figure 11.

The mymodel:StoreURI element that is used for addressing it part of the model of the pet store, which the invoker is expected to understand (assuming the invoker is the same as the recipient of the response).

In this scenario, making the invoker aware of the addressing mechanism does not add much coupling because the invoker is already

Figure 10. Request message to retrieve the state

```
<soap:Envelope>
  <soap:Header>
    <wsa:MessageId>http://foo.com/m1</wsa:MessageId>
    <wsa:To>http://hp.com/JPetStoreManager</wsa:To>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/Get
  </wsa:Action>
  <mymodel:StoreURI>http://www.PetsRUs.com/PetStore/index.jsp
</ mymodel:StoreURI>
  </soap:Header>
<soap:Body/>
```

Figure 11. Reply message to state request

```
<soap:Envelope>
  <soap:Header>
    <wsa:MessageId>http://hp.com/m1</wsa:MessageId>
    <wsa:RelatesTo>http://foo.com/m1</wsa:RelatesTo>
    <wsa:To>http://foo.com/Service1</wsa:To>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse
  </wsa:Action>
  </soap:Header>
<soap:Body>
  <mymodel:PetStore>
    <mymodel:StoreURI>http://www.PetsRUs.com/PetStore/index.jsp</ mymodel:StoreURI>
    <mymodel:StoreOwner>Joe Pet</mymodel:StoreOwner>
    <mymodel:HasSecuredPaymentService>true</ mymodel:HasSecuredPaymentService>
    <mymodel:Payment Accepted>
      <mymodel:PaymentType>Credit</mymodel:PaymentType>
      <mymodel:PaymentType>Debit</mymodel:PaymentType>
    </mymodel:Payment Accepted>
  </mymodel:PetStore>
</soap:Body>
</soap:Envelope>
```

assumed to understand the semantics of the elements used for addressing. The only additional assumption is that the invoker needs to know which elements of the model can be used for addressing.

So, in this scenario, the cost of making addressing information transparent to the invoker is low. The main benefit of doing so is that with this information in hand, the invoker does not need to first retrieve an EPR for the resource. It has, without the need for such an EPR, all the information needed to create a complete message, thus saving the need to send a query to a registry (which requires finding a registry in the first place). Note that this is only a saving if the invoker already knows the address of the endpoint and if it doesn't need access to metadata potentially contained in the EPR. As a result, this is not useful in a federated scenario, where the address to which the message is sent varies. In addition, a provider might optimize its dispatching mechanism by choosing specific ways to build the EPR. If the invoker is allowed to use model elements to address the resource, it effectively prevents the resource manageability provider from being able to optimize its dispatching mechanism. Another limitation is that the invoker might not be the same as the recipient of the response and in this case the invoker might not be expected to understand the resource model.

EVALUATION

In this section we evaluate our solution by presenting our experience in developing the solution and evaluating the scalability of our prototype.

Experience in Global Service on PlanetLab

We have built our prototype on PlanetLab to manage several instances of JPetStore services deployed globally. Scale, complexity, and dynamism of the PlanetLab environment resembles the systems of future. PlanetLab is an evolving research testbed, and so are the next generation distributed services. Because we based our design on standard solutions for the various aspects of the system, we were able to build our prototype in less than two weeks. Our experiments consisted of deploying the JPetStore application on 50–100 nodes. On each node, the JPetStore was customized based on its geographic location. This customization required initializing each JPetStore database with its respective products. The entire deployment process was visualized with a tool that showed dots on the screen as the deployment completed or failed. During our experiments with this prototype, we note the following interesting events that took place:

- **Dynamism:** PlanetLab nodes constantly went up and down. Our initial list of nodes to be deployed during the experiment had 100 nodes. However, a large number of those nodes went down during the course of the experiment, and as a result, our number of nodes shrank from 100 to 36, then to 22 and then to 12. In less than 36 hours, our setup was reduced to one tenth of its original size.
- **Dependencies:** Just before starting one of the runs of our experiment, the configuration of the eventing infrastructure was modified. This change was not formally captured by our system, and as a result, the configuration changes made to it were not propagated to our management services and the experiment broke unexpectedly.
- **Varying load:** The PlanetLab network and nodes were highly loaded leading to unpredictable service response times. Our management service is neither currently handling adaptation to such changing conditions, nor the underlying infrastructure provides any sort of resource guarantees. As a result, the time for the deployment events to get propagated through the communication infrastructure to the visualization tool was much poorer than expected, with some events taking more than five minutes to complete.

Thus, even though the deployment service code was functioning correctly and executed the deployment of JPetStore, the interesting characteristics of the computing and service environment caused a disruption in the experiments. The lesson learned is that the management service needs an adaptability layer, which can adapt to the PlanetLab environment conditions. We also require a more structured formal representation (i.e., model) of the overall system, including characteristics such as ranges of expected performance and response times, to allow the management system to deal with expected behavior, and to identify anomalous behavior.

Quantitative Evaluation

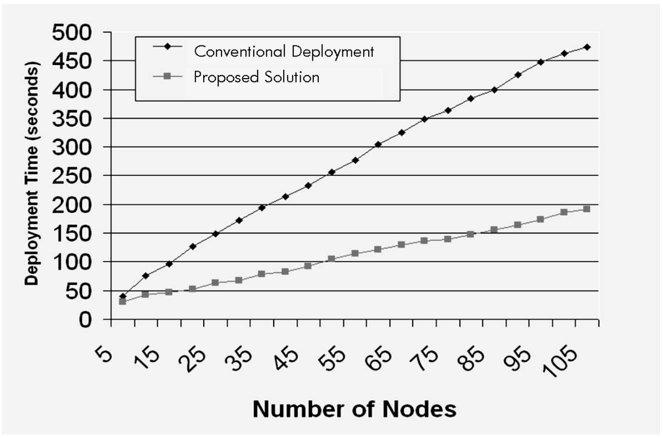
We conducted several additional experiments to validate the scalability of our Web-services–based JPetStore deployment. We conducted a scale experiment for deploying JPetStore on up to 105 PlanetLab nodes (see Figure 12). Nodes were chosen at random from around the world. The infrastructure on each node for our deployment service included Apache Tomcat Web Server, Apache Axis Web Service Container and our deployment Web service component. We created a work flow for deploying JPetStore (see Figure 12). We then conducted a scale experiment for the deployment and collected deployment time. Our measurement infrastructure consists of a process monitoring the start and finish times of each JPetStore deployment. The data is then analyzed to calculate the average deployment time with increasing scale. As seen in Figure 12, compared to a conventional deployment service, the slope of our deployment service is much lesser resulting in the average deployment time to increase slowly as the number of nodes increases. For example, a doubling of deployment nodes from 25 to 50 results in only 67% increase in deployment time, tripling of nodes from 25 to 75 results in 120% increase in deployment time, and quadrupling the nodes from 25 to 100 results in only 195% increase in deployment time. Although there is potential for further improvement in scalability, the current results look promising. Although we would like to conduct scale experiments with hundreds if not thousands of nodes, we expect to see a similar pattern for the graph.

LESSONS LEARNED

We have learned the following lessons while developing our approach to scalable management:

- *Planetary scale requires careful interaction between applications, service oriented architectures, and the management stack.* At the scale of millions of nodes, transparently extending architectures designed for

Figure 12. Scaling Web services deployment



small scale will not work. Our preliminary experience indicates that such applications must be designed with planetary scale in mind. Rather than transparently hiding scalability, in many cases it must be exposed, by explicitly designing for federation and distribution where needed.

- *There exist new and different challenges for scalable management with respect to reliability and availability.* Because of the complexity of services, the dependency matching and redeployment of services becomes a critical part of the system. Consequently, separating it from the deployment system (and thus decoupling it from the health of the deployment system) is essential even if it introduces additional problems in maintaining this additional state.
- *Models are only as good as what we want to do with them.* We are relying on the use of models. However, distributed models also pose new challenges for maintaining their consistent state across distributed service deployments. Therefore, the models are not contributing anything in their own right, the key benefit is in the use of models in a manner that serves the given purpose best. This usually means making decision based on incomplete knowledge of the system.
- *Transparency of the WSDM interfaces.* We have used scalable eventing for communication. WSDM was designed with point-to-point management, whereas the eventing was designed for one-to-many communication. WSDM interfaces accommodated for it without any changes to existing design and implementation.
- *Move the BPEL workflow to target machines.* In our current implementation, the deployment workflow specified in the BPEL language is processed by a centralized BPEL workflow engine hosted at the deployment server. Such a design enables processing of cross-node dependencies at a single workflow engine. However, such a centralized orchestration has the limitations of scale. There is a need to partition and distribute the BPEL workflow description to workflow engines on the target machines. Challenges exist to achieve decentralization, such as determining the partition boundaries, and co-ordination among the distributed workflow engines.

RELATED WORK

The related work falls into categories of deployment, model-based automation, and workflows. In the area of deployment, several tools exist. The Deployme system for package management and deployment supports creation of the package, distribution, installation, and deleting old unused packages from remote hosts (Oppenheim & McCormick, 2000). Magee et al. describe CONIC, a language specifically designed for system description, construction, and evolution (Magee, Kramer, & Sloman, 1989). Cfengine provides an autonomous agent and a middle- to high-level policy language for building expert systems that administrate and configure large computer systems (Burgess, 1995). A number of other tools are surveyed in Anderson, Goldsack, & Paterson (2003).

Existing management solutions similarly address functionalities in other areas of our interest; for example, adaptation to failures and to performance violations; HP OpenView (www.managementsoftware.hp.com); IBM Tivoli (www.tivoli.com); Computer Associates Unicenter (http://www3.ca.com/solutions/solution.asp?id=315). The effectiveness of these traditional solutions in large distributed systems is significantly reduced by a number of properties of these solutions. These are centralized control, tight coupling, nonadaptivity, semiautomation. Furthermore, these solutions do not adequately address the needs and characteristics of large-scale distributed services.

We base our work on standards evolving in SOA. SOA represents a tie between various areas, such as Grid computing, autonomic computing, and enterprise computing, by enabling underlying mechanisms for implementing policies and controls for these different domains. A number of projects use workflows for orchestrating tasks in large scale dynamic environments, such as Krammer, Bolcer, and Taylor (1998) and Vidal, Buhler, and Stahl (2004). Our work has a lot of similarities with all above areas, however, our primary focus is on very large scale, distributed services. Such services could be running on, for example, geographically distributed data centers. Managing these services require *loose coupling* of the management stack, *decentralization*, and *dealing with incomplete knowledge*. Our management system leverages scalable technologies; for example, publish-subscribe, decentralized agents and control, and extends them further to the next level of very large scale global services. We provide solutions for deployment, health, and adaptation for services lifecycle management. Furthermore, we provide higher level abstractions for service and system descriptions through languages and models, which aid in formally capturing the complex needs of emerging services.

SUMMARY AND FUTURE WORK

In this article we have described an approach for managing planetary-scale services. Our approach is based on the use of models and standards. We have demonstrated a use case and then presented our solution to it, followed by an initial evaluation. We claim that adopting this approach will enable easier management of global services and reduce development and adoption barriers. In summary, it will reduce the total cost of ownership of large computer systems running global scale services.

Areas of future work include extensions to the proposed management services in terms of functionality and scalability. In particular, the deployment service can be enhanced to handle failures and exceptions during orchestration process, the federated models can be integrated with adaptation services, and the security framework can be extended to include accountability. Further improvements to scalability can be achieved through decentralization (e.g., using distributed workflows) and loose coupling (e.g., using Web services over publish-subscribe eventing).

ACKNOWLEDGMENT

Parts of this work were conducted in the broader context of the Scalable Management project, with Robert Adams and Paul Brett.

REFERENCES

Adams, R., Brett, P., Iyer, S., Milojicic, D., Rafaeli, S., & Talwar, V. (2005). Scalable management—Technologies for management of large-scale, distributed systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC),* Seattle, WA.

Anderson, P., Goldsack, P., & Paterson, J. (2003). SmartFrog meets LCFG: Autonomous reconfiguration with central policy control. In *Proceedings of the 17th Large Installation System Administration Conference (LISA),* San Diego, CA.

Brett, P., Knauerhase, R., Bowman, M., Adams, R., Nataraj, A., Sedayao, J., et al. (2004). A shared global event propagation system to enable next generation distributed services. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS),* San Francisco, CA.

Burgess, M. (1995). A site configuration engine. *USENIX Computing Systems, 8*(3).

Foster, I., Kesselman, C., Nick, J., & Tuecke, S. (2002). The physiology of the grid: An open grid services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum.*

Gilat, D., Landau, A., & Sela, A. (2004). *Autonomic self-optimization according to business objectives.* Proceedings of the International Conference on Autonomic Computing (ICAC), New York, NY.

Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., & Toft, P. (2003). SmartFrog: Configuration and automatic ignition of distributed applications. In *Proceedings of the HP OpenView University Association Conference.*

Housley, R., Ford, W., Polk, W., & Solo, D. (1999), Internet X.509 public key infrastructure certificate and CRL profile. *RFC 2459.*

Huhns, M. N., & Singh, M. P. (2005). Service-oriented computing: Key concepts and principles. *IEEE Internet Computing, 9*(1), 75-81.

Kandlur, D., & Killela, J. (Guest eds.). (2004). Utility computing. *IBM Systems Journal, 43*(1).

Krammer., P., Bolcer, G. A., Taylor, R. N., & Hitomi, A. S. (1998). Supporting distributed workflow using HTTP. In *Proceedings of the Fifth International Conference on the Software Process*, Lisle, IL.

Magee, J., Kramer. J., & Sloman, M. (1989). Constructing distributed systems in conic. *IEEE Transactions on Software Engineering, 15*(6), 663–675.

Oppenheim, K., & MCormick, P. (2000). Deployme: Tellme's package management and deployment system. In *Proceedings of the Usenix 14th Large Installation System Administration Conference (LISA),* New Orleans, LA.

Peterson, L., Anderson, T., Culler, D., & Roscoe, T. (2002). A blueprint for introducing disruptive technology. In *Proceedings of ACM HotNets-I Workshop*, Princeton, NJ.

Rafaeli, S., & Hutchison, D. (2003), A survey of key management for secure group communication, *ACM Computer Surveys, 35*(3), 309-329.

Talwar, V., Milojicic, D., Wu, Q., Pu, C., Yan, W., & Jung, G. (2005). Approaches for service deployment. *IEEE Internet Computing, 9*(2), 70-80.

Wang, Y. M., Verbowski, C., Dunagan, Y., Chen, Y., Helen J. Wang, et al. (2003). STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th Large Installation System Administration Conference (LISA),* San Diego, CA.

Vidal, J. M., Buhler, P., & Stahl, C. (2004). Multiagent systems with workflows. *IEEE Internet Computing, 8*(1), 76–82

Wilkes, J., Mogul, J., & Suermondt, J. (2004). Utilification. In *Proceedings of the ACM SIGOPS European Workshop*, Leuven, Belgium.

ENDNOTE

[1] Planetlab (Peterson et al., 2002) is a research testbed consisting of nodes spread all over the globe. All of the nodes run a common software package that includes a Linux-based operating system and support for distributed virtualization.

William Vambenepe is an HP Distinguished Technologist in the OpenView Office of the CTO where he is one of the architects of the technical strategy for HP OpenView. In addition to driving technical alignment with partners and key customers, he oversees the standards strategy for OpenView.

Vanish Talwar is a researcher in the Enterprise Systems and Software Lab at Hewlett-Packard Laboratories. His technical interests include distributed systems, operating systems, and computer networks, with a focus on management technologies. He received his MS and PhD degrees in computer science from the University of Illinois at Urbana Champaign (UIUC) in 2001 and 2006 respectively. He is the recipient of the David J. Kuck Best Masters Thesis award in the Department of Computer Science, UIUC, and is an elected member of Phi Kappa Phi and Sigma Xi. He is also a member of the ACM, IEEE, and USENIX.

Sandro Rafaeli is a senior software engineer and researcher for HP Brazil. He has obtained his PhD in Computer Science at the University of Lancaster, England. Before joining HP, he has participated in several research projects in the area of distributed systems funded by the European Community.

Bryan Murray is a member of the Architecture team in HP's Advanced Technology Office and interacts with architects and developers throughout HP's OpenView Division. Bryan represents HP in several OASIS technical committees and DMTF working groups. Bryan has co-authored specifications used as a basis for the specifications published by these groups and has been editor of some of the specifications.

Dejan Milojicic is a senior researcher and a project manager at HP Labs. He has worked in the area of operating systems and distributed systems for more than 20 years. He has been the program chair of the IEEE Agent Systems and Applications Symposium (ASA/MA'99) and of the first ACM/IEEE/USENIX Workshop on Industrial Experiences with System Software (WIESS'2000). Dr. Milojicic published in many journals and at various events. He is currently on the editorial board of IEEE Distributed Systems Online. He has been engaged in various standardization bodies and helped standardize OMG MASIF and more recently works on standardizing SmartFrog configuration framework. He is a member of the ACM, IEEE, and USENIX. He received his BSc and MSc from University of Belgrade and his PhD from University of Kaiserslautern.

Subu Iyer is a systems software engineer and researcher at HP Labs, Palo Alto. He joined DEC Network Systems Lab in 1997 where he worked on collecting and analyzing performance data from a large cluster of machines on DEC's Palo Alto Research Gateway. Over the years, Subu has worked on projects in the areas of distributed computing, performance monitoring and telepresence. His current work is on scalable adaptive performance monitoring.

Keith I. Farkas is a member of the VirtualCenter R&D team at VMware. His research interests include manageability of distributed applications, distributed resource management, mobile computing and applications, microprocessor design, and power- and energy- efficient solutions for pocket and server computer systems. He is on the editorial board for IEEE Pervasive Computing Magazine. He received his PhD from the University of Toronto. He is a member of the IEEE and the ACM.

Martin Arlitt is a researcher in the Enterprise Software and Systems Laboratory at HP Labs.