

Scalable Management

Technologies for Management of Large-Scale, Distributed Systems

Robert Adams[†], Paul Brett[†], Subu Iyer, Dejan Milojicic, Sandro Rafaeli, Vanish Talwar

HP Labs, Intel[†]

[firstname.lastname]@[hp, intel].com

Abstract

Modern computing environments, such as enterprise data centers, Grids, and PlanetLab, introduce distributed services to address scalability, locality, and reliability. Web Services (WS), in particular, improve decoupling, decentralization, and autonomicity within distributed systems. Unfortunately, scale and decentralization introduce additional problems in distributed services management, such as deployment, monitoring, and lifecycle maintenance.

In this paper, we propose a new approach to management of large scale distributed services, based on three artifacts: scalable publish-subscribe eventing, scalable WS-based deployment, and model-based management. We demonstrate that these techniques improve the manageability of services. In this way we enable service developers to focus on the development of service functionality rather than on management features.

1 Introduction

Contemporary computing systems are increasing in scale and broad deployment across the globe. This is true for enterprise, scientific systems, as well as consumer space. Traditional centralized enterprise data centers are expanding into dozens of geographically dispersed data centers. Remote operations are contending with even more management complexity while also dealing with the emergence of hundreds of so called closet computers in small branch offices and home offices. Leveraging computation or data assets in Grid [1] or PlanetLab [2] environments pose similar requirements. As applications and services move out of the datacenter and into distributed installations, a new class of applications and services are coming about which are large-scale, geographically distributed, shared, and heterogeneous.

This has dramatically changed the design assumptions for such systems and applications. *Scalability* is not limited any more by physical or administrative boundaries—systems span the globe and cross organiza-

tions. *Availability* is not driven only by private networks and corporate policies—many systems are connected over wide area network and outside of a given administrative domain. This results in a significant *dynamism* in terms of unexpected loads, rebooting and upgrading machines and services.

We claim that as systems continue to grow in size and wide-area deployment, traditional management approaches, such as those currently used by OpenView [3], Tivoli[®] [4], and Unicenter[®] [5], will become less effective. The management systems are moving towards service oriented architectures [16] as demonstrated by the recent standards, such as WSDM [17] and WS-Management [18]. But, scalability, availability and dynamism create additional requirements.

The features we consider essential are *loose coupling* of the management stack (communication, deployment services, and model-based automation), *decentralization* (distribution, no central point of management), and as a result of previous two, *dealing with incomplete knowledge*.

To demonstrate the utility of these features to scaled management, we created a scalable, decentralized, distributed service provisioning and management system which includes three significant artifacts: Planetary Scale Event Propagation and Router (*PsEPR*, pronounced “pepper”) [6], an infrastructure for scalable, publish-subscribe eventing which scales significantly better than point-to-point or hierarchical topologies; *WS-based service deployment* tool [7] which decouples deployment specification from the dependencies and component models; and finally *model-based automation* which enables changes to the design of the system at run time, enabling a higher degree of automation.

These three artifacts enable future application developers to more easily to design, develop, and manage a distributed applications that have no deployment or management center (decentralized), that are geo-

graphically disperse and that adapt to changing resource availability and workload. We have built these three artifacts on the PlanetLab test bed [2], which has been used for the last several years for deployment and testing of this class of applications. Some key learnings from running very large scale applications and services on PlanetLab align very well with our goals of decoupling, decentralization, and dynamism.

Even though we do not explicitly address self-* characteristics in this paper, scalable management is closely related to much of the work in autonomic computing. The two areas share a number of required and recommended behaviors for autonomic computing [8]. For example, in order to accommodate scale, management must be fully automated, i.e. self-managed; it must handle problems locally whenever possible (i.e. the impact of a change in an area should not impact other services at the global scale); and scalable services' behaviors and relationships must be managed so that they meet service level agreements. Furthermore in order to accomplish scalable management, underlying systems must implement these design patterns such as self-configuration, self-healing, and self-optimization. While in this paper we do not explicitly address these patterns, the topics of manageability automation, adaptation, performance, and dependencies are critical for large scale autonomic systems. In addition, scalable management requirements such as decoupling, decentralization, and dealing with incomplete knowledge, are also features of autonomic systems.

1.1 Motivating Scenarios

To illuminate the required features of scalable management, we present three motivating scenarios: global service health, inventory; and plug-in.

Global Service Health. Consider a service that provides some functionality to people or computers all over the world, runs 24/7, is hosted at hundreds of locations that are geographically separated and which is made up of many interacting components. Somehow, the service must decide which hosts to run on, allocate the resources for on those hosts and then install and configure itself on those hosts. The set of hosts will be constantly changing because of hardware failures, network failures, purposeful reconfiguration of the hardware or network and because of malicious activity. Additionally, the number of hosts required by the service can change because of work load or new business requirements. Also, the number of separate components of the application can be constantly changing and thus the installation and reconfiguration process is continuous.

Running on multiple, geographically disperse locations has the advantage that the service has increased immunity to failure and attack. But, from a management point of view, it is hard to know if the service is running correctly. This service demonstrates the extremes of decentralization and decoupling. So, besides the problems of deploying and configuring a decentralized application, there are problems of management and control.

Global Service Inventory. Consider an installation of computers that spans the globe. This could be all the desktop computers in a multi-national corporation or all of the blades in a collection of data centers that have been geographically located around the world. Monitoring and controlling all of these computers becomes difficult at some scale. Manual and semi-automatic management of the systems will seek solutions like running similar applications on all of the computers and limiting the variations in hardware configurations. However, data centers will only grow and the number of client computers will only increase. This growth will require automated management and control. Because of unreliable monitoring systems and the network, the management and control feature will need to run in multiple locations.

Usual solutions are to centralize management and to build hierarchies of managers—clients are managed by low level management systems and these low-level managers are managed by other managers and these managers are controlled by a central manager. It is easy to see that these layers create more complexity and more things to manage. Additionally, managing the managers has the same problems as managing the low level computers. In a summary, this scenario requires scalable communication that connects managers and other scalable components and automated management/control interface.

Global Service Plug-in. Consider a service that uses several services to perform its function. If there is a need to install this service in a new environment, a number of services that this service depends on may already be running, but some may not. Of the already running services, some of them may be the right version, but the others may be obsolete and a new version needs to be installed.

Furthermore, the running services, with the right version, need to be verified for correctness of operation prior to installing a new service. Correctness also includes the service level agreements that need to be guaranteed for the composite service. Once everything

is verified and all dependencies have been resolved, the new service needs to be “plugged-in” into existing services, by dynamically connecting new service with existing services. In a summary, this scenario requires service discovery or the updated model of the system, service health monitoring, and loose and recoverable connection between services

The remainder of the paper is organized in the following manner: Section 2 motivates the paper with an analysis of scalability and complexity. In Section 3 we present related work in the area. Section 4 describes architecture, design, and implementation of the three artifacts. In Section 5 we evaluate performance of our solutions, followed by lessons learned in Section 6. Summary and future work are presented in Section 7.

2 .Dealing with Scale and Complexity

To further motivate the need to deal with scalability and complexity, we have performed two experiments.

Discreet event simulations of centralized, hierarchical and decentralized control structures were constructed in order to predict the behavior at large scale of these control structures. These simulations, based on published measurements of the global Internet for response times [9] and packet loss [10], simulate the effects of TCP delays and losses on the planetary scale command and control structure implemented on top of TCP. Figure 1 shows the results obtained for a resource constraint of 100 simultaneous connections at each node, with a 3.5% probability of packet loss and with the TCP recovery strategy [11]. Additionally, these simulations included introducing delays due to resource constraints at each node and connection failures

Centralized management and control of applications were seen to suffer from significant performance degradation at scale, due to resource constraints and error rates on globally distributed networks.

Hierarchical topologies improve the control and management scalability significantly, but reach limits on very large networks due to the cumulative effect of network failure as the tree depth increases.

Shared routing overlay networks such as PsEPR provide better performance on globally distributed networks by amortizing the overhead of maintaining optimum routing architectures over many applications. Additionally, the use of a common command and control structure provides improved resiliency to real-world latencies and error rates.

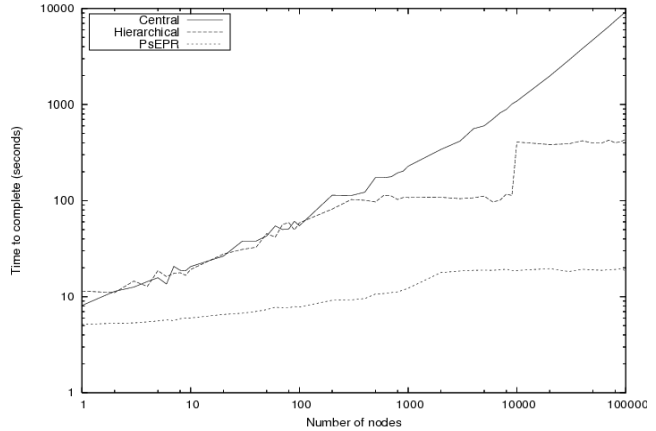


Figure 1. Comparison of managing approaches: centralized, hierarchical (spanning tree) and publish-subscribe (simulated).

Relatively speaking, improvements in latency significantly lag improvements in bandwidth [12]. The impact of this is the decreasing efficiency of static, hierarchical communication structures and the increasing performance of decentralized, dynamic structures.

In the second experiment, we have compared the number of required changes as a result of a reconfiguration or failure. We have evaluated the number of changes as a function of service complexity and scale. We looked at system changes in response to dynamic events for a simple application - a JPetStore application, a medium complex application - a local content provider and a complex application - an airline reservation system running in multiple countries in different languages. Our analysis of system changes in response to dynamic events exhibits the challenges faced in designing automated management services with an ever increasing complexity of systems.

The dynamic events introduced are those of *application server failures*, and *addition of new application servers*. The higher level dynamic events result in several subsequent changes within the system. This is attributed to the complex interdependencies that exists among the system components. As can be seen from the graph presented in Figure 2, the number of needed system changes grow exponentially with an increase in complexity. The problem becomes even more challenging in very large scale systems wherein management services are decentralized and have to make decisions based on incomplete knowledge. The graphs illustrate a very critical problem, that of designing automated management services that can deal with an ever increasing complexity of the system.

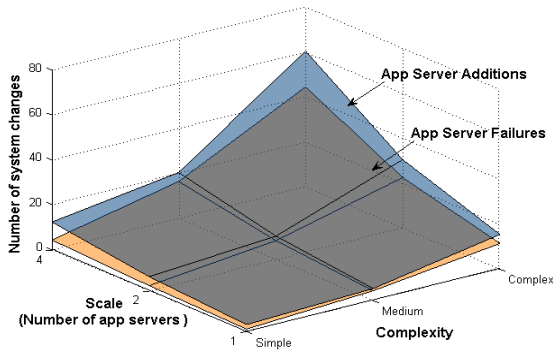


Figure 2. A Number of Complex Service Changes as a function of scale & Complexity. The scale is shown with respect to number of app servers which are affected by the dynamic event.

Our approach to handling this complexity is to model applications and services and multiple loosely connected components. The information model infrastructure captures the current state of the system. Whenever dynamic events occur, they are propagated to the information models. The information models are then reasoned upon by adaptation management services to determine the low level system changes that needs to be implemented in the system. These changes are then executed within the system. Other management services that rely on system knowledge, for example, monitoring services, only need to refer to the updated information model to obtain updated current system state.

3 Related Work

Our work on scalable management draws a lot of similarities with work in many areas. While we leverage the existing experience, we are different in that our primary focus is on the very large scale, global services. In particular, we base our work on service oriented architectures, but in order to accomplish the scale, we are required to adopt autonomic techniques.

PsEPR is similar in concept to publish/subscribe systems. These range from Java Message Service [22] to TIBCO Corporation's Rendezvous [23]. PsEPR differs from these by dynamically creating communication points dynamically so event senders and receivers have minimal dependencies. PsEPR's overlay routing is also opaque thus allowing services to adapt to it's structure -- for instance, moving computation "close" to data sources. This sort of messaging structure is also being explored in Astrolabe[24]. There are also many other examples of publish subscribe co-ordination and communication efforts [25, 26, 27, 28].

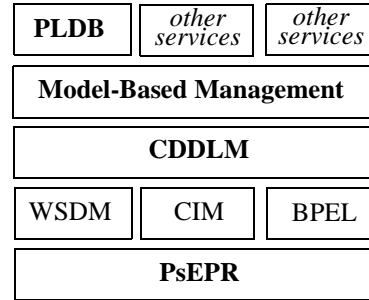


Figure 3. High Level Architecture

In terms of related work in the area of application management systems, several deployment tools exist. Deploye system for package management and deployment supports creation of the package, distribution, installation, and deleting old unused packages from remote hosts [30]. Kramer et al. describe CONIC, a language specifically designed for system description, construction, and evolution [31]. Cfengine provides an autonomous agent and a middle to high level policy language for building expert systems which administrate and configure large computer systems[32].

Existing management solutions similarly address functionalities in other areas of our interest, e.g., adaptation to failures and to performance violations ([3], [4], [5]). 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, non-adaptivity, semi-automation. Furthermore, these solutions do not adequately address the needs and characteristics of large-scale distributed services. Most of the tools do not by themselves provide complete lifecycle management capability necessary in large dynamic systems such as Planetlab.

In contrast, we are designing our management system by leveraging scalable technologies, some of which are mentioned in this section, e.g., publish-subscribe, decentralized agents and control, decentralized decision making, and extending them further to the next level of very large scale global services. We provide solutions for deployment, eventing, and adaptation for services lifecycle management. We also propose higher level abstractions for service and system descriptions through languages and models, which aid in formally capturing the complex needs of emerging services.

4 Architecture, Design, and Implementation

The architecture of our system is presented in Figure 3. It consists of the PsEPR [6], on top of which three

industry standard packages are running: OASIS *Web Services Distributed Management* (WSDM) defines management interfaces and schemas [17], DMTF *Common Information Model* (CIM) describes how information and state is modeled [19], and *Business Process Execution Language* (BPEL) supports the workflow for services [20]. On top of these components, the deployment service is running as an implementation of the GGF *Configuration Description, Deployment, and Lifecycle Management* (CDDL) standard [21]. On top of the stack is the automation engine that automates deployment and management of the whole stack. As an example of a managed application we are using Planet-Lab Data Base (PLDB). In the rest of the section we describe in more detail the PsEPR, deployment, and automation layers.

4.1 PsEPR/PLDB

To build loosely-coupled, distributed applications, we created an event-based communication system named Planetary Scale Event Propagation and Router (PsEPR). For communication of monitoring and control information, PsEPR creates an overlay network for the distribution of XML messages from a source to one or more receivers (See Figure 4).

Our experience with building and managing a large, disturbed service [2] lead us to conclude that loose coupling among components (within or between distributed services) is necessary for robust distribution. Specifically, communication between virtual endpoints, where those endpoints can move (or be transparently redirected) as necessary because of the ever changing characteristics of communication bandwidth and availability. Our definition of “loose-coupling” includes:

- location independence of senders and receivers—'location' both in network address space and in physical space;
- service independence—neither the provider nor the user of a service needs to know of the existence of the other;
- state independence—reliability or delivery guarantees are not required;
- connection flexibility—one-to-many and many-to-many communication is easy.

Thus, PsEPR creates an overlay network on the existing Internet that efficiently moves event messages from clients sending the events to clients who have asked to receive the messages.

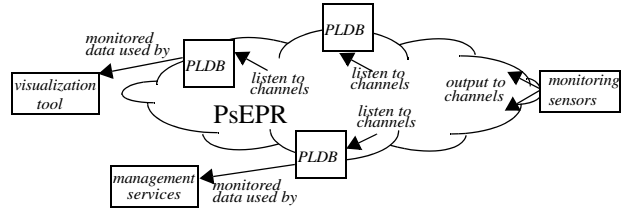


Figure 4. PsEPR Infrastructure and PLDB

The PsEPR communication model sends XML formatted messages over named “channels”. Channels are hierarchically named with channels having sub-channels, etc. A client authenticates itself to PsEPR and sends event messages to any channel. To receive events, a client requests a “lease” on a particular channel -- a request to receive messages of a particular type from a channel and its sub channels.

This is similar to a publish/subscribe system where event senders create the messages and receivers 'subscribe' to the messages they wish to hear. For instance, a client on host named “x.example.com” could send heartbeat messages on a channel:

```
con = new PsEPRConnection(credentials);
con.send(heartbeatEvent,
"/example.com/heartbeat/x.example.com/");
```

One or more receivers could be listening to all heartbeat messages for this class of clients:

```
c = new PsEPRConnection(credentials);
les = c.getLease("/example.com/heartbeat/", 120, typeHeartbeatEvent);
event = les.receiveEvent();
```

In this simple example, the receiver has asked for all heartbeat events on the “/example.com/heartbeat/” channel and all of its sub-channels for the next 120 seconds. Since the sender is sending events addressed to a sub channel of that lease, the receiver will see it along with events sent on channels of other hosts. If the receiver only wanted events from the one host, it could subscribe to that particular sub-channel.

Internally, PsEPR is made up of Routers which accept events from Clients, route the events among the Routers and deliver the events to other Clients based on routing tables. These routing tables are built by a Registry service which runs parallel with each Router. The Registry service processes the 'lease' requests and thus knows who is listening for events on channels.

The Registries communicate among themselves to pass information on where leases are originating. In this way, the Routers implement an ever changing tree from senders and receivers. The current implementation uses

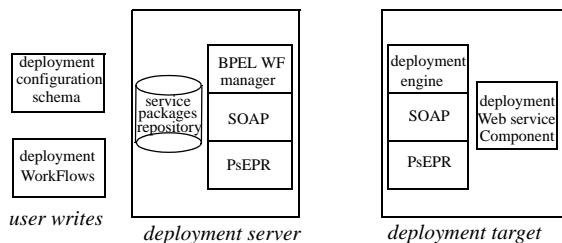


Figure 5. Deployment Components

a simple forwarding table, but we shall enhance infrastructure with optimizing route calculations.

Measurements of PsEPR show that, while PsEPR is less efficient at point-to-point communication and high volume transfers, its flexibility makes adaptation of changing service configuration simple. Loose-coupling of service components, in the ways that PeEPR makes available, creates more reliable and scalable services and systems.

One service that has been implemented on top of PsEPR is PlanetLab Database (“PLDB”). Since PsEPR events are transient (they are lost if not received), PLDB is a service that recalls past events that appeared on certain channels.

Clients have been put on all PlanetLab nodes that output onto PsEPR channels information about the state of the node. Any program wishing to know the current state of a node could listen to that node's channel. But, these clients only put out a tuple of information when the value of that tuple changes. This necessitates some way of finding the last event sent out. Rather than creating a query-like communication to a tuple sender (it's not the sender's problem that the receiver hasn't been listening forever), requests for events from the past was generalized into a service (“PLDB”) which listens to channels and remembers the last values for tuples.

PLDB is made up of multiple Supervisors who each manage a collection of Monitors. Each individual Monitor listens to one channel and collects and stores tuples that are seen on that channel. The Monitor can also generate tuples based on requests it sees on the channel -- some Client wishing to see a past tuple value sends a request event and the Monitor replays a version of the tuples with time information.

Each Supervisor who is managing a group of Monitors also listens to the traffic on a set of channels and independently evaluates the number of Monitors that are operating on a channel. If there are too few monitors running, it creates a Monitor for that channel. If there are too many Monitors on a channel, the Supervisor can terminate one of it's Monitors. Heuristics around the

number of Monitors on the channel, the timing of creation and destruction and geographical load balancing creates an ecosystem of Monitors listening to a set of channels.

4.2 Deployment

We have built a system for deploying large scale decentralized services within wide area infrastructures. The functional requirements for this deployment system are to perform the installation, configuration, activation, and re-configuration of services. It addresses the challenges of scalable performance, high reliability, and fast recovery time in response to dynamic faults and workload variations. The design of our deployment system builds on the lessons and experiences gained from the SmartFrog Project at HP [33] and the CDDLML working group at GGF [21].

Figure 5 shows the conceptual view of the deployment system components. The key aspects are: *decentralized management components* which describe the deployment actions of a service, a *deployment configuration schema* that describes the configuration information needed during deployment, *deployment workflows* that compose the management components, and a *decoupled communication mechanism* based on SOAP-PsEPR. The deployment system components are conceptually distributed among a deployment server and a deployment target machines.

Application providers or writers typically specify the deployment details of the application service, e.g. steps needed to install the software, the list of dependent packages and services, in README files and manuals. Given the scale and complexity of the systems, there is a need to express the deployment information in a more structured and machine-readable manner, so as to be able to automate the complete deployment process in a repeatable way. Given an application service, an administrator using our proposed approach describes the logic for installation, configuration, and activation of the service as Java methods of a management component. The management components extend well-defined deployment interfaces. The code snippet below shows an example management component. These management components are then distributed to all of the deployment targets.

```
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);
}
}
}

```

At the time of deployment, the deployment administrator expresses the configuration information needed during the deployment process in a well-defined deployment configuration schema.

The administrator also describes the various dependencies that the service has with other distributed services and applications as a BPEL workflow. In this workflow, the deployer maps the dependency requirements that the application service provider has specified to the actual instances of the packages and services within the system. For example, an application writer specifies that this application needs an Oracle DB. The deployer maps this requirement to an actual Oracle DB available somewhere and specifies that in the BPEL workflow. The BPEL workflow appears as a composition of the management components.

```

<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>

```

The BPEL workflow is provided to a BPEL process manager responsible for orchestrating the deployment actions in accordance with the specified workflow. The BPEL process manager communicates with a *deployment engine* that exists on all of the deployment targets. The deployment engine on a deployment target node is responsible for receiving and processing all of the deployment requests given to that deployment target node. It parses the requests sent through a BPEL engine, locates the appropriate management component responsible for a request, and then invokes the appropriate methods on that component. That method is responsible for executing the deployment actions for the service.

A workflow for a typical complex service would involve multiple management components, some of

which are invoked and executed in parallel and others in sequence.

We are implementing our deployment service on the basis of the design presented above. Our initial use case scenario is the deployment of PLDB. The software package for PLDB consists of a tar file for the core software, and a set of dependent libraries that the software needs. The dependencies are expressed as BPEL workflows, and supplied to a ActiveBPEL workflow engine. We are creating a library of commonly used deployment components. For example, we have a RPMInstaller component, a RSyncDownloader component, a Notifier component among others. These components are being written in Java.

These generic components are then reused for the design of the deployment components written for PLDB application. An early version of the deployment engine has been developed in Java and hosted as a web service within a Tomcat-Axis container on every deployment target (managed client).

A new transport mechanism has been integrated in the Axis stack to enable handling SOAP calls over PsEPR. Extending the Axis stack is just a matter of extending the BasicHandler class. We have called our PsEPR handler PsEPRSender:

```

public void invoke(MessageContext msgContext) throws AxisFault {
    SoapPayload myPA = new SoapPayload(
        msgContext.getRequestMessage().getSOAPPartAsString() );
    PsEPREvent myEV = new PsEPREvent();
    myEV.setPayload(myPA);
    PsEPRConnection myConn = new PsEPRConnection(credentials);
    myConn.sendEvent(myEV);
}

```

The PsEPR enabled client is required to set the new transport to the Call object. The Axis engine finds the link between PsEPRTransport and PsEPRSender in the client-config.wsdd created from the XML file below:

```

<deployment name="pepr" xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<handler name="PsEPRSender"
type="java:soap.pepr.PsEPRSender" />
<transport name="PsEPRTransport" pivot="PsEPRSender" />
</deployment>

```

The key benefits of our proposed design are (i) decentralized deployment process through management components and BPEL dependency specification for scalable and reliable deployment, (ii) standards based, high-level interaction (SOAP, WSDL and BPEL) for increased inter operability and decreased recovery time, (iii) workflow description expressiveness through BPEL language. Overall, we provide automation and lowered management costs through our system.

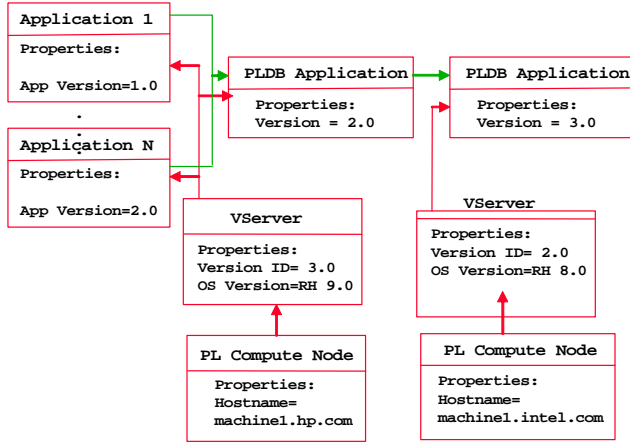


Figure 6. Conceptual partial view of the PLDB and the underlying infrastructure model.

4.3 Model-based Automation

Large scale application services and systems provide challenges towards the design of automated management systems. For example, a management service responsible for self-adaptation to dynamic changes is required to deal with information and processes that are heterogeneous, of large size, and dynamic. The problem gets compounded further as complexity of the system increases, mandating a knowledge of intricate administrator learnings during key management decisions.

We propose a model-based design of automated management services to deal with the challenges mentioned above. Figure 6 shows the conceptual view of the components of our design. In such a design, information models present a structured, formal representation of the information about the IT system and services. The information model provides a set of well defined modeling classes and schemas to represent information about hardware elements, software services, their relationships and associated constraints. An example of such schemas and specifications are those defined by CIM. We build upon the CIM schemas within our prototype implementation.

The models are stored in model repositories. For scalable management, our design proposes a federation of distributed model repositories, each individual repository captures the local system information. A well defined model object manager and interfaces exist to access the information contained in the repositories.

Distributed model repositories present several challenges. First, an appropriate partitioning of the system information is needed which accounts for locality of ref-

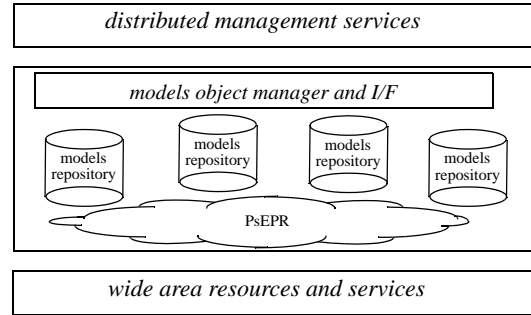


Figure 7. Model Based Automation Component

erence, and semantics of the stored information. Second, models need to be kept consistent across the system as a whole and have to deal with partial updates.

Thirdly, the model object managers must support a scalable distributed query mechanism. Further, the model repositories themselves need to be self-adapting to changes, e.g. faults, occurring in the system. As an ongoing effort, we are designing solutions addressing these challenges. We are also extending the model management subsystem to provide support for histories, transactions, and multiple consistency levels.

In a typical usage of models, schemas for the system under consideration are designed. The designed schemas support multiple levels of abstraction of system information. Instances of the designed model are subsequently created and stored in the distributed model repositories. They are initialized with information on current state of the local system for which they are responsible. Thereafter, the instance models are continuously updated to reflect new states of the system.

The model repositories thus capture the complex system information in a structured and distributed manner, and they together provide a near-real time view of the entire system (see Figure 7). Our model-based decentralized management services rely on the information captured by models during their decision making processes. At any given time, a particular component of the decentralized management service selects a subset of model repositories to obtain current system information. The choice of this subset is statistical and depends on various performance and locality properties.

Once the subset of model repositories is selected, the management service chooses the level of abstraction within the model that is most appropriate to its needs. The decision making is then done using the incomplete system knowledge.

We present two examples of automated management services to illustrate the design. First, consider an

adaptation service, that determines the set of adaptation actions to be taken in response to dynamic events. Whenever an event occurs in the system, it is propagated to the model repositories, and the model instances are updated to reflect the event. The models at this point have captured the complex current state of the system in a structured meaningful manner. The adaptation service applies reasoning on this structured information, and determines the set of low level system wide changes e.g. the set of redeployment actions, that need to be implemented in the system.

Next, consider a resource allocation management service that processes monitoring data collected in a distributed system. The service needs to know information about the monitoring collectors/reporters etc. This is a challenge in a complex, dynamic, and heterogeneous system consisting of several hundreds of computing elements, each with their own collecting and reporting infrastructure. With our approach, this complex information is captured in models. The management service is designed to only refer to the models to obtain the information. The model is continuously updated to reflect the new system state even in the presence of dynamic system changes.

We are using OpenPegasus software as the basic infrastructure for storing and retrieving CIM-based models. The implementation of distributed model repositories is currently a work in progress. Our future and ongoing effort also includes prototyping an adaptation service based on our model-based approach within Planetlab environment.

5 Performance Evaluation

We have performed a few experiments in order to verify the scalability of our management system. We have performed measurements for PsEPR, WS-based deployment, models, and the PlanetLab Data Base (PLDB).

We were interested in comparing the performance of two types of Web service communication: the synchronous SOAP over HTTP with the asynchronous SOAP over PsEPR. The Web service used in these experiments is a dummy service, which simulates the execution of a deployment operation. We have used Axis as the Web service container for both HTTP and PsEPR scenarios. Axis is easily integrated into Tomcat and it also allows the transport layer to be changed. The HTTP Web service was made available through Tomcat and we have written a PsEPR server to substitute Tomcat for handling PsEPR-SOAP requests.

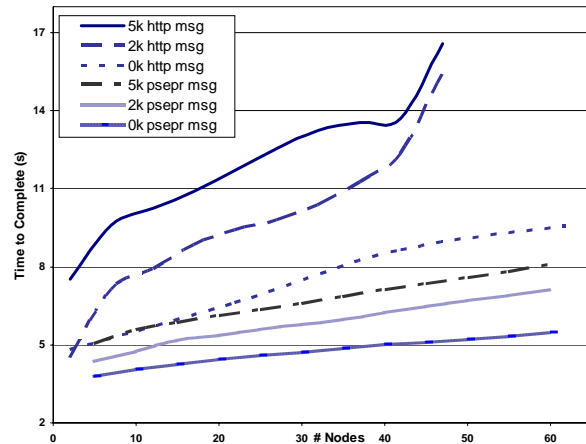


Figure 8. SOAP over HTTP vs. SOAP over PsEPR.

The PsEPR server gets a lease on a channel and retrieves the SOAP payload received in PsEPR events. The SOAP payload is handed to an Axis engine. The Web service response is returned by the Axis engine to the PsEPR server and then it is sent back to the originator of the call.

We have two clients for originating requests, one executes synchronous communication (HTTP), which means the client sends the request and receives the response in the same thread, and the other executes asynchronous communication (PsEPR), which means the request is sent in a thread and the response is received in an independent thread.

Our network of Web services runs on over fifty nodes of Planet Lab, each one running a few instance of the Tomcat and the PsEPR server totaling around sixty-five instances of each server. In each experiment (HTTP and PsEPR), we have measured the time taken to execute the calls to a set of Web services running on a number of nodes. A call consists of sending a request and waiting for a response either synchronously or asynchronously. We run the same experiments increasing the size of the payloads. As described in Section 4.2, deployment configuration schemas are distributed to deployment targets. We wanted to verify what is the impact of using larger SOAP envelopes (simulating more complex schemas) on the performance of Web service calls. Figure 8 demonstrates the increase on the time span of a deployment operation when the number of nodes involved in the operation increases and also the size of the payload is increased.

We conducted a few experiments to find out the feasibility of using WSDM MUWS for our scalable management solution. We compared Web Services

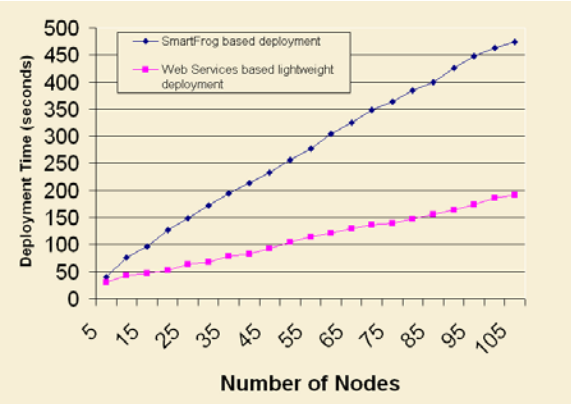


Figure 10. SmartFrog vs Web Services based deployment compared on increasing number of targets.

using Axis as the Service Container against MUWS. Our experiment mainly focused on finding out how much overhead both systems incurred when the scale increased. We ran experiments comprising of 2 to 87 nodes on PlanetLab. Figure 9 shows how the two solutions compared among each other with increasing scale. The experiment consisted of making a synchronous call to a Web Service running using WSDL and another one running using MUWS and waiting for a response. As you can see from the figure, WSDM has more overhead than the WSDL based approach, but the overhead is not significant. So, we believe that the advantage of using WSDM MUWS outweighs the disadvantage of the overhead.

We conducted several experiments that compare SmartFrog based deployment against our Web Services based deployment solution. The experiments consisted of deploying PLDB in a series of PlanetLab nodes. The nodes were chosen from a geographically dispersed set of locations around the world. We varied the number of deployment nodes from 1 to 105 and found that the Web Services based light weight solution consistently outper-

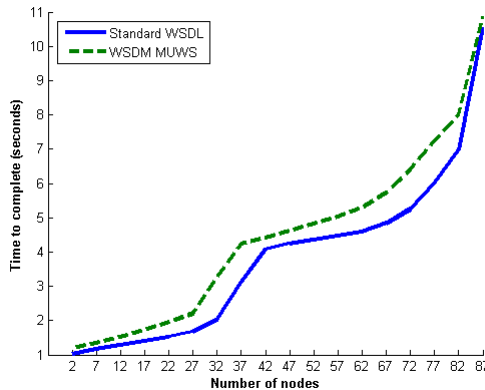


Figure 9. Scaling management using WSDM.

formed the SmartFrog based deployment solution in terms of deployment time when the scale increased. The results of this test are shown in Figure 10. The experiments cemented our belief that, although Web Service solution is perceived to perform slower and consume more memory and CPU, the differences are less marked in realistic applications [34]. Moreover, we believe that some of the known advantages of the Web Services based solution such as inter operability and extendability simply outweigh the drawbacks. The purpose of the experiments was not to prove that the Web Services based solution is better than the SmartFrog based solution, but rather how Web Services based deployment is a viable solution for large scale deployment. It doesn't seem fair to compare SmartFrog which is feature rich against our light weight Web Services based solution. Nonetheless, the results are encouraging and show a pattern that we expect to see in terms of scalability and extensibility. In the future, we plan to improve our solution to have a richer functionality. We also plan to replace the underlying communications stack from HTTP to PsEPr. We believe that using PsEPr for deployment will improve scalability and reliability while the use of Web Services will improve interoperability.

One of the services we have constructed based on our eventing, deployment, and management principles is the PlanetLab database service, a tuple-store service providing management information for PlanetLab. Numerous PLDB monitors running on PlanetLab observe properties like load average, currently installed packages, and kernel checksums which are transmitted via PsEPr to any listening services. PLDB achieves robustness, reliability, and high availability through service replication. A management supervisor monitors health of the tuple-store service and dynamically starts and stops local monitors on a per channel basis in order to maintain robustness, reliability and availability goals. Figure 11 shows a set of PLDB monitors running on a group of channels. When all the monitors on a single node are killed, other node supervisors detect the reduction in redundancy on a per channel basis, and automatically create new channel monitors to restore the service to it's design parameters.

6 Lessons Learned

In this section we summarize some lessons learned while exploring scalable management of global services.

- There are trade-offs between performance and reliability for traditional point to point communication v.

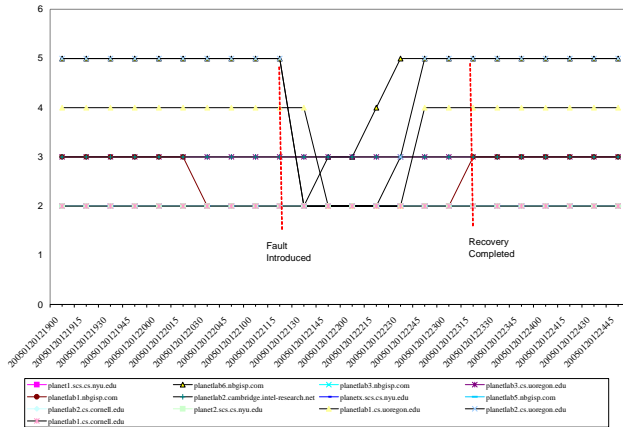


Figure 11. Flexing PLDBs

loosely-coupled publish-subscribe. Our preliminary simulation as well as real system performance measurement indicated the scalability benefits of the latter. PsEPR enables decoupling at the lowest layer. PLDB's ability to use multiple channel monitors showed that loose-coupling (client's finding each other in PsEPR channels) creates easy and transparent reliability from the client perspective.

- Decoupling at the communication layer is not enough. We also need decoupling higher in the stack. This requires an event-driven programming model in the design of management services. While a fully decentralized and decoupled service is ideal to handle scale, it opens a problem in managing it. We thus end up with building a decentralized management solution to manage a decentralized application service, and then we need another decentralized management solution to manage decentralized management service, and so on. There is still an open question in how to fine tune the balance between decentralization and ease of management
- We decoupled the expression of dependencies from the component model, such as in SmartFrog. This enabled us to reason and manage the dependencies through workflows. However, this introduced the need to manage the expressed dependencies (install, update as they change, etc.). There exists trade-off between improved expressiveness and development time (e.g. of workflows, language). As we develop higher level of abstractions, such as expressing dependencies for deployment, it enables more degrees of run-time design changes.
- The proposed solution to address the complexity problem is to build solutions that capture complexity in a structured manner, based on models. This way the "effort" needed in dealing with complexity is being

shifted from "runtime" to "development time". However, there is a trade-off that exists in this shift in terms of maintenance cost, software development effort, and disruption to existing systems design.

- There is a need to architect for global services. Services for reliable global operation are different from applications built for the machine room. Their requirements are different and rely primarily on scalability, complexity, dealing with incomplete knowledge.

7 Summary and Future Work

We have presented a new approach for scalable management, based on decoupling, decentralization, and dealing with incomplete knowledge. We demonstrated design and implementation of three system components that contribute to the architecture of scalable management: a scalable publish-subscribe evening, WS-based deployment, and a model-based management. We have evaluated performance of these components in terms of scalability. All these features are critical for autonomous systems of future.

In the future, we are going to explore extensions to the WSDM interface for scalable management (multicast management channels) and workflows for managing multiple interdependent components. We are also going to add more features to our management components and make it available as a toolkit to PlanetLab community. We plan to capture their experience of researchers in the form of best practices for scalable management. One area that we specifically want to focus on is policies and best practices for management of large scale globally distributed services. Once we have the basic scalable management infrastructure in place and it is used by the PlanetLab users, we shall be able to experiment with different policies and capture and derive the best practices.

Acknowledgments

We are indebted to Martin Arlitt, Greg Astfalk, Sujata Banerjee, Ira Cohen, Puneet Sharma, and William Vambenepe for reviewing the paper. Their comments significantly improved the contents and presentation of the paper. Mic Bowman and Patrick McGeer provided original support and ideas for pursuing this effort. Dongyan Xu shepherded our paper through the review and submission process.

References

- [1] I. Foster et al, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", Open Grid Service Infrastructure WG, Global Grid

- Forum, June 22, 2002. <http://www.globus.org/research/papers/ogsa.pdf>
- [2] L. Peterson, et al., "A Blueprint for Introducing Disruptive Technology into the Internet", Proceedings of the First ACM Workshop on Hot Topics in Networking (HotNets), October 2002.
- [3] HP OpenView <http://www.managementsoftware.hp.com/>.
- [4] IBM Tivoli, <http://www.tivoli.com/>.
- [5] Computer Associates Unicenter, <http://www3.ca.com/solutions/solution.asp?id=315>.
- [6] P. Brett, et al., "A Shared Global Event Propagation System to Enable Next Generation Distributed Services", WORLDS'04: First Workshop on Real, Large Distributed Systems, San Francisco, CA, December 2004.
- [7] V. Talwar et al., "Approaches for Service Deployment", IEEE Internet Computing, vol. 9, no. 2, pp. 70-80, March-April 2005.
- [8] S. White, et al., "An Architectural Approach to Autonomic Computing," Proceedings of the International Conference on Autonomic Computing, pp 2-9, May 2004, New York, NY, USA.
- [9] Jeremy Stribling, "All Pairs Pings for PlanetLab" http://www.pdos.lcs.mit.edu/~strib/pl_app
- [10] "Internet Traffic Report - Global Packet Loss" <http://www.internettrafficreport.com/30day.htm>
- [11] V. Paxson, RFC-2988: Computing TCP's Retransmission Timer, <http://rfc.net/rfc2988.html>, November 2000
- [12] David A Patterson, "Latency Lags Bandwidth", Communications of the ACM, October 2004, pp71-75
- [13] J. Dunagan, et al., "Towards A Self-Managing Software Patching Process Using Black-Box Persistent-State Manifests," Proceedings of the International Conference on Autonomic Computing, pp 106-113, May 2004, New York, NY, USA.
- [14] G. Chen and D. Kotz, "Dependency Management in Distributed Settings," Proceedings of the International Conference on Autonomic Computing, pp 272-273, May 2004, New York, NY, USA.
- [15] S. Aiber et al., "Autonomic Self-Optimization According to Business Objectives," Proceedings of the International Conference on Autonomic Computing, pp 206-213, May 2004, New York, NY, USA.
- [16] M.N. Huhns and M.P. Singh, "Service-Oriented Computing: Key Concepts and Principles," IEEE Internet Computing, vol. 9, no. 1, 2005, pp. 75-81.
- [17] OASIS WSDM WG Charter <http://www.oasis-open.org/committees/wsdm/charter.php>
- [18] WS-Management - <http://msdn.microsoft.com/ws/2005/02/ws-management/>
- [19] DMTF CIM, <http://www.dmtf.org/standards/cim/>
- [20] OASIS BPEL Working Group Charter: <http://www.oasis-open.org/committees/wsbpel/charter.php>
- [21] CDDLM Charter, <https://forge.gridforum.org/projects/cddlm-wg>
- [22] M. Happner et al., "Java Message Service 1.1", <http://java.sun.com/products/jms/docs.html>.
- [23] TIBCO Corp., "TIBCO Rendezvous", http://www.tibco.com/software/enterprise_backbone/rendezvous.jsp
- [24] R. van Renesse, K. Birman, and W. Vogels. "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining", ACM Transactions on Computer Systems, Vol. 21, No. 2, pp. 164-206, May 2003.
- [25] P.R. Pietzuch "Hermes: A Scalable Event-Based Middleware". Ph.D. thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.
- [26] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford, "T Spaces", IBM Systems Journal, Vol. 37, No. 3, pp. 454-474, 1998.
- [27] M. Narayanan, et. al., "Approaches to asynchronous Web services", <http://www-106.ibm.com/developerworks/web-services/library/ws-asoper/>
- [28] IBM Corporation. MQSeries: An Introduction to Messaging and Queuing. Technical Report GC33-0805-01, IBM Corporation, June 1995. <http://ftp.software.ibm.com/software/mqseries/pdf/horaa101.pdf>.
- [29] T. De Wolf, et al., "Towards Autonomic Computing: Agent-based Modeling, Dynamical Systems Analysis, and Decentralized Control", Proc 1st Int'l Workshop on Autonomic Computing Principles and Architectures 2003.
- [30] Oppenheim, K., and McCormick, P., .Deployme: Tellme.s Package Management and Deployment System., Proceedings of the Usenix IVth LISA Conference, December 2000, New Orleans, pp187-196
- [31] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing Distributed Systems in Conic. IEEE Transactions on Software Engineering, 15(6):663--675, June 1989
- [32] Mark Burgess, "A Site Configuration Engine", USENIX Computing Systems, Vol8, no 3, 1995, <http://www.cfengine.org>
- [33] P. Goldsack, et al., "Configuration and Automatic Ignition of Distributed Applications", 2003 HP Openview University Association conference.
- [34] N.A.B. Gray, "Comparison of Web Services, Java-RMI, and CORBA service implementations", Proc. of Fifth Australasian Workshop on Software and System Architectures (ASWEC), April 2004.