RC23950 (W0605-011) May 2, 2006
Computer Science

IBM Research Report

Experience with Collaborating Managers: Node Group Manager and Provisioning Manager

Ian Whalley, Asser Tantawi, Malgorzata Steinder, Mike Spreitzer, Giovanni Pacifici, Rajarshi Das, David M. Chess
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Experience with Collaborating Managers: Node Group Manager and Provisioning Manager

Ian Whalley, Asser Tantawi, Malgorzata Steinder, Mike Spreitzer,
Giovanni Pacifici, Rajarshi Das, David M. Chess
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
{inw,tantawi,steinder,mspreitz,giovanni,rajarshi,chess}@us.ibm.com

Abstract

This paper presents an autonomic system in which two managers with different responsibilities collaborate to achieve an overall objective within a cluster of server computers. The first, a node group manager, uses modeling and optimization algorithms to allocate server processes and individual requests among a set of server machines grouped into node groups, and also estimates its ability to fulfill its service-level objectives as a function of the number of server machines available in each node group. The second, a provisioning manager, consumes these estimates from one or more node group managers, and uses them to allocate machines to node groups over a longer timescale. We describe the operation of both managers and the information that flows between them, and present the results of some experiments demonstrating the effectiveness of our technique. Furthermore, we relate our architecture to a general autonomic computing architecture based on self-managing resources and patterns of inter-resource collaboration, and to emerging standards in the area of distributed manageability. We also discuss some of the issues involved in incorporating our implementation into existing products in the short term, and describe a number of further directions for this research.

1. Introduction

The vision of autonomic computing [1] is of computing systems that manage themselves to a far greater extent than they do today. To achieve this vision, we believe that interacting sets of individual computing elements must work together to regulate and adapt their own behavior in response to widely changing conditions, with only high-level direction from humans.

In [2] we describe an architecture for autonomic systems, in which self-managing resources (there called "autonomic elements") regulate their own behaviors according to policies and interact via agreements and relationships, in order to achieve overall system self-management.

New architectures are seldom implemented in a single step; we do not expect the world to simply toss out existing systems and replace them with new self-managing ones built according to this architecture. The adoption of self-management techniques and architecture will generally be a gradual process. As more autonomic capabilities are incorporated into existing system management technologies, opportunities will arise to introduce the features of the architecture gradually into the systems that include them.

In this paper we describe an IT management system in which two management components, each with some autonomic characteristics, collaborate to achieve a level of overall system self-management within a data center. This paper is an extended version of [3], updated to describe some more recent algorithms and results, and to relate our work to emerging manageability standards.

The first manager in our system, a node group manager implemented in a middleware application server, uses modeling and optimization algorithms to allocate server processes and individual requests among a set of server machines ("nodes") grouped into node groups. It also estimates its ability to fulfill its service-level objectives as a function of the number of nodes potentially available to each node group. We describe the most important algorithms used in making these calculations, and the information that the node group manager surfaces to the other components of the system.

All of the nodes in a given node group share a set of properties, such as installed operating system, network connectivity, support libraries, and so on. The node

group manager is responsible for directing and balancing the traffic within each node group, and for allocating server processes to the nodes within the group, but it is not able to carry out the provisioning actions necessary to move nodes from one node group to another. Additionally, the node group manager has only a local view of the set of node groups that it is responsible for; it cannot make higher-level decisions about the allocation of nodes between the demands of its own node groups and those of other processes in the larger data center.

The second manager we describe, a provisioning manager, implements another layer of management above the node group manager. It has the knowledge necessary to move nodes from one node group to another through potentially time-consuming provisioning actions, and it can balance the competing demands of multiple managers operating at the level of the node group manager. (These may be instances of the node group manager, or of other managers that can provide the required performance estimates.) On the other hand, the provisioning manager does not have the node group manager's real-time knowledge of the traffic within each node group.

The two management layers are thus complementary; the two managers collaborate in that the estimates produced by the node group manager allow the provisioning manager to more effectively allocate the resources that it provisions, and the actions of the provisioning manager give the node group manager more servers to work with (when it has sufficient need for them).

In operation, the provisioning manager consumes performance estimates from one or more node group managers (and other managers operating at the same level), and uses them to allocate nodes to node groups over a longer timescale, using the knowledge mentioned above. In the body of this paper, we briefly outline the function of the provisioning manager, and describe the semantics of the information that it exchanges with the node group manager.

Finally, we analyze this system in terms of the autonomic architecture described in [2], highlighting those features that bear on architectural solutions to the challenges of self-management.

The remainder of the paper is organized as follows. In Section 2 we describe the algorithms used by the node group manager for flow control and process placement, and for collaboration with the provisioning manager; we cover these algorithms in some detail, to give a concrete example of the calculations that occur within autonomic management software, and how some of the results of these are exposed to other parts of the system. In Section 3 we outline the function of the provisioning manager, and in Section 4 we describe the interactions between the two.

In Section 5 we describe a simple scenario that exercises the autonomic functions of the components, and present the results of experiments that demonstrate system self-management capabilities in that scenario. In Section 6 we relate the system as currently implemented to the autonomic architecture and to emerging manageability standards. In Section 7 we outline some directions for future work, and in Section 8 we conclude.

2. The application server

The node group manager used in this system is implemented within a piece of application server middleware ("ASM"): specifically, a version of IBM's WebSphere Extended Deployment product. This product has a wide variety of features and functions; for the purposes of this paper we consider only the subset of those features described below.

In the ASM, the relationship between applications and server machines (called *nodes*) where the applications can execute is expressed in terms of *node groups*. A node group is a set of nodes that have capabilities matching requirements of some application. Server machines within a node group may be heterogeneous, and node groups may overlap. An application is mapped to a single node group. For each application mapped to a node group, a *dynamic cluster* ("DC") is created, which is a set of application-server instances that run on one or more nodes within the node group. The ASM dynamically manages resources used by an application by controlling the request flow into the system, by load balancing requests across dynamic-cluster members, and by dynamically modifying the size and placement of dynamic clusters.

Figure 1 illustrates the topology of the system described here; the ASM is in the upper left. A user request first arrives at one of M on-demand routers (ODRs). Each ODR contains several logical gateways (drawn as ovals in the figure), each of which is dedicated to a particular dynamic cluster (see below) and contains queues, a scheduler, and a load balancer. We use the variable p to index into the set P of ODRs. The ODR schedules and directs the request to a server process on one of N server nodes, under the supervision of multiple control loops. (For brevity, only a single node group is illustrated.) Node's server processes are drawn as circles. In this paper, we use the variable n to index into the set N of nodes in the system.

Incoming requests are categorized in the ODR, according to administrator-specified criteria, into *service classes*. We use the variable c to index into the set C of

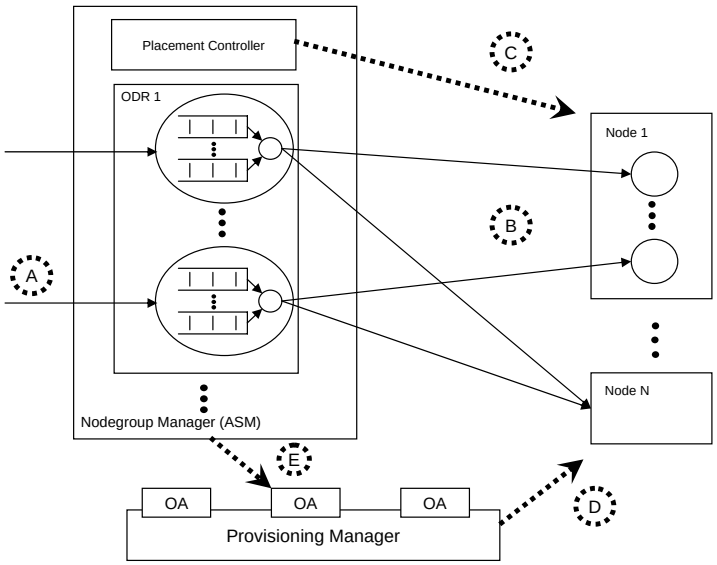

Figure 1. Overall system structure (described in detail in the text). Incoming requests (A) are queued, and dispatched (B) to application servers by a flow controller and workload manager (not shown). On a longer timescale, the placement controller determines (C) which application servers should run in which nodes. In the longest timescale, the provisioning manager determines (D) which nodes to assign to which node groups by consulting its objective analyzers, which receive estimates (E) from the nodegroup manager (for simplicity only one node group is shown here).

service classes. For each service class, the administrator specifies a service level objective by setting a threshold and an importance value as described in Section 2.2.

At any given time, each node in the system is running the server processes corresponding to some subset of the DCs; a node is said to be "in" all of the DCs whose server processes are currently running on it. The set of DCs that any particular node is in can change over time, under the control of the placement algorithm described in Section 2.1.

When a request arrives at an ODR, it is classified and put into a queue specific to the request's service class and the DC that will serve it. Each ODR limits the number of requests it has running in each DC at a given time. When the appropriate DC is under the request limit and one or more requests are available in the relevant queues, one of those queues is picked using a weighted round robin discipline, and a request is dequeued from that queue. A server in the DC is chosen by a load balancer that uses another weighted round robin scheme, and the request is forwarded to the chosen server. The server processes the request and returns a reply to the ODR, which forwards the reply to the original client.

The ASM includes three feedback control loops for managing performance. The first such loop is the placement control loop, which determines the DCs associated with each node (that is, which application server processes run on each node). The second control loop, the flow manager, manages the competition between service classes and between DCs, taking the current placement as a given. This controller computes the DC concurrency limits and the round-robin weights used in dequeuing. The flow manager also provides information for use by other parts of the system. For example, it produces demand signals sent to the placement controller, as described below. The third controller, called dWLM, adjusts the load balancing weights used in request routing (its function is not relevant to this paper, and we will not mention it again).

The following subsections discuss the placement control loop and the flow manager in more detail.

2.1. Placement

The placement controller periodically computes a new desired instance placement, and then stops and starts application server processes as necessary to put the new desired instance placement into effect.

We use the term *instance placement* for a matrix

I of 0s and 1s, where $I_{d,n}$ is 1 if and only if node n is currently in DC d (that is, it reflects whether or not a server process for that DC is running on that node).

The choice of an instance placement matrix is defined as a two-dimensional bin-packing problem. Our approach to the problem is essentially that described in [4]. Each node n is configured with a load-dependent capacity ("speed") Ω_n and load-independent capacity Γ_n. Each DC d is configured with a load-independent demand γ_d, and assigned a load-dependent demand ("speed request") ω_d by the flow manager (which provides new values each control cycle). An instance placement matrix I solves the problem when there exists a load placement matrix L, with cells containing non-negative real numbers, where:

$$\forall n : \forall d : (R_{d,n} = \text{false}) \quad \Rightarrow \quad I_{d,n} = 0 \qquad (1)$$

$$\forall n : \forall d : (I_{d,n} = 0) \quad \Rightarrow \quad L_{d,n} = 0 \qquad (2)$$

$$\forall n : \Gamma_n \quad \geq \quad \sum_d \gamma_d I_{d,n} \qquad (3)$$

$$\forall n : \Omega_n \quad \geq \quad \sum_d L_{d,n} \qquad (4)$$

$$\forall d : \omega_d \quad = \quad \sum_n L_{d,n} \qquad (5)$$

In Eq. (1), R represents the allocation restriction matrix defined by node to node group mapping. $R_{d,n}$ is equal to *true* only if node n is a member of the node group that dynamic cluster d is mapped to.

Before computing a new desired instance placement, the controller first tests whether the current instance placement solves the problem; if so, no change is done. Otherwise, a new instance placement is computed. That computation uses a heuristic that tries to minimize the average case amount of change from the current instance placement (because changes are relatively expensive).

2.2. The flow manager

The algorithms used in the flow manager are based on those described in [5]. The flow manager produces control and inter-controller signals intended to maximize a system-wide objective function U. That objective function is determined from goals set by the administrator, as follows:

$$U = \min_{p \in P} \min_{d \in D} \min_{c \in C} U_c(\bar{P}_{p,d,c}) \; . \qquad (6)$$

Here $\bar{P}_{p,d,c}$ is the expected performance result for the *flow* identified by ODR p, DC d, and service class c. Each service class c is defined to have a particular objective function $U_c()$ of a particular kind of performance result metric. Currently two kinds of performance result metrics are supported: (1) average response time, and (2) percentage of response times above a certain given amount of time. For each service class c the administrator may specify an associated threshold value τ_c and *importance* value z_c, which parameterize the class' objective function. We use class objective functions of the following form:

$$U_c(p) = \left(\frac{\tau_c - p}{\tau_c} \right) \cdot \begin{cases} 1 & \text{if} \quad p \leq \tau_c \\ \frac{100 - z_c}{99} & \text{if} \quad p \geq \tau_c \end{cases} \qquad (7)$$

Thus, the class objective values are modified stretch factors, and the system-wide objective is to even out the modified stretch factors.

The expected performance result for a flow is a function of the resources allocated to that flow. That function is based on a performance model, and the model's parameters are extracted on-line from observations of the system in operation.

The resources allocated to a flow are *seats*; that is, the number of "slots" available for requests in the various DCs, as constrained by each DC's concurrency limit.

For each flow p,d,c, the flow manager uses the performance model to derive the number of seats needed to meet the flow's SLO; we write $o^*_{p,d,c}$ for that number of seats.

To compute the control signals at a given time, taking an instance placment I as a given, the flow manager first makes a nominal allocation, to each server process s and flow $p,d(s),c$ that hits that server process, of a non-negative integer number $\breve{o}_{p,c,s}$ of seats that flow may occupy on that server process. The resource constraint is this:

$$\forall n : \sum_{s \in S(n)} \sum_{p,c} \kappa_{p,d(s),c} \, \breve{o}_{p,c,s} \leq \hat{\rho} \, \Omega_n \qquad (8)$$

Here $S(n)$ is the set of server processes running on node n, and $\hat{\rho}$ is a configured CPU utilization target. This resource constraint says that each seat takes up a certain amount of the node's computing power, and that amount depends on the flow. The coefficients $\kappa_{p,d,c}$ are computed by an auxiliary component known as the Work Profiler. The nominal allocations are not enforced directly; rather, through certain summations and other processing, the actual control signals — the dequeuing weights and DC concurrency limits — are derived from the nominal allocations.

When computing the demand signals for placement, the flow manager does a similar optimization — but does not take the current instance placement as a constraint. Let $D(n)$ denote the set of all DCs that are mapped to node groups that include node n, i.e., DCs

such that $R_{d,n} = \text{true}$. In this case, the controller allocates a non-negative real number $o_{p,d,c}$ of seats subject to the resource constraint:

$$\forall n : \sum_{d \in D(n)} \sum_{p,c} \kappa_{p,d(s),c} \check{o}_{p,c,s} \leq \hat{\rho}\, \Omega_n \qquad (9)$$

Note that in this case we are not limited to the DCs that the node is currently in, but rather consider allocating to all eligible DCs. Both optimizations are done every control cycle. Let us add the subscript i to index over control cycle iterations, starting with 1.

The flow manager calculates the demands signals for the placement controller as follows. Using the placement-unconstrained allocations, the flow manager computes in each cycle, for each ODR p and DC d, a speed request precursor

$$\check{\omega}_{p,d,i} = \sum_{c} \kappa_{p,d,c,i} \min(o_{p,d,c,i}, \bar{l}_{p,d,c,i} + \bar{o}_{p,d,c,i}) \qquad (10)$$

where $\bar{l}_{p,d,c,i}$ is the recent average queue length and $\bar{o}_{p,d,c,i}$ is the recent average seat occupancy for flow p,d,c at iteration i. Clipping by the sum of those two avoids requesting speed that is unlikely to actually be utilized.

Next, the speed request precursors are smoothed, using a moving averaging technique with weights that decrease with age, to produce values we write as $\bar{\omega}_{p,d,i}$. This process also produces standard deviation values, which we write as $\nu_{p,d,i}$. This smoothing compensates for the noise in the system and the mismatch in control cycles (the placement controller runs on a different — typically longer — control cycle than the flow manager).

Finally, whenever the time comes to produce the demand signal for DC d, the flow manager sums across ODRs and adds in a multiple of the root-mean-square variance:

$$\omega_d = \left(\sum_{p} \bar{\omega}_{p,d,i} \right) + 2 \sqrt{\sum_{p} \nu_{p,d,i}^2} \qquad (11)$$

Including the variance makes some allowance for the demand fluctuations expected before the next placement controller run.

3. The provisioning manager

The provisioning manager (PM) used in the autonomic system described here is IBM Tivoli Intelligent Orchestrator (TIO) [6] [7]. This provisioning manager has a large number of features, but this paper will focus only on those relevant to the discussion at hand. The PM dynamically reallocates server machines among a set of application environments.

The key concepts used by the PM to model the managed system are *application environment*, *cluster*, and *resource pool*. An application environment is a modeling construct used to assign SLA objectives to a particular workload. It is composed of all resources, i.e., server machines that are used to serve this workload. These resources are divided among clusters. To avoid confusion with application-server clusters, we will call them provisioning-manager clusters (PMC). A PMC is a set of server machines that are identical with respect to their power, OS, installed software stack, and network connectivity. For multi-tiered application environments, PMCs usually serve different application tiers. Neither PMCs nor application environments overlap, i.e., a server machine may not be a member of two PMCs at a time and a PMC cannot belong to two application environments at a time.

A PMC is associated with exactly one resource pool, which is a homogeneous set of server machines. Multiple PMCs may be associated with a single resource pool. The PM automatically moves servers from a resource pool to an associated PMC by executing a workflow, which appropriately configures the server and installs needed software. Similarly, the PM automatically removes a server from a PMC and moves it back to a resource pool by running a different workflow, which reverses the configuration and uninstalls the software.

The decision whether or not to add or remove a server machine is made by an optimizer, which allocates server machines from a resource pool among the associated PMCs based on the SLA objectives defined for the corresponding application environments and the current performance of the corresponding workloads. The objective of the optimizer is to minimize the probability that an SLA objective is violated for any managed application environment. Hence, for any application environment, the PM must be able to estimate the probability that its SLA is violated for any given allocation of resources.

Application environments managed by the PM may greatly differ with respect to the type of workload, SLA metrics used, and their internal resource management logic. Therefore, it is impossible for the PM to accurately model the performance of all these environments. Instead, the PM relies on application environments themselves to provide the probability of SLA violation.

In order to ensure the flexibility required to manage a wide variey of subsystems, the PM provides an interface by which evaluation logic designed explicitly for

a given application can be 'plugged in'. The decision logic module is called an 'Objective Analyzer' (OA). Such Objective Analyzers are also discussed in [8].

In general, each application environment (within the provisioning manager) can have an Objective Analyzer associated with it. The Objective Analyzer encapsulates performance modelling, monitoring, and analysis needed to calulate the probability of violating the SLA for a given allocation of server machines to a given PMC within the application environment. The Objective Analyzer is invoked by the provisioning manager on a cycle — the current version of the provisioning manager runs on a 30 second decision cycle. When invoked, the Objective Analyzer constructs a 'probability of breach' (PoB) curve for each PMC in the application environemnt — a sample PoB curve is shown in Figure 2.

The PoB curve shows, for any possible number of server machines n assigned to the PMC in question, the probability $P(n)$ of violating the SLA of the associated application environment assuming that n server machines are assigned to the PMC. In practice, the PoB curve is implemented as an interface, and the provisioning manager calls back into that interface for various values of n (that is to say, various numbers of server machines) and is provided, in response, with the appropriate $P(n)$.

The two horizontal lines shown on Figure 2 delineate the probability of breach 'target zone' — in the current version of the PM, $p_1 = 0.65$, and $p_2 = 0.35$. As long as the probability of breach for a particular PMC remains within the target zone, the provisioning manager will not change the number of server machines allocated to that PMC, unless another PMC is starved of resources. If the probability of breach for a given PMC is above the target zone the provisioning manager is more likely to give that PMC additional nodes, and if it is below the target zone it is a likely candidate to have nodes taken from it.

4. Collaboration between ASM and PM

The collaboration between ASM controllers and PM requires two problems to be solved: modeling ASM concepts in the system model of PM, and integrating the control logic of the two environments. These issues are discussed in this section.

4.1. Mapping ASM concepts to the provisioning manager

One of the challenges in a multi-manager environment is reconciling the potentially different ways that the various managers look at the world. As discussed in Section 2, the ASM administrator creates node groups and allocates the available server machines among them. Application server processes are grouped into dynamic clusters, whose distribution is bounded by the node groups.

The most natural way of translating ASM concepts to PM concepts is by mapping ASM node groups to PM clusters (PMCs). With such a mapping, PMCs provide a boundary for the deployment of application, which is desirable from a user perspective. However, this mapping has quite obvious problems:

- Since node groups may overlap, to correctly represent this concept in the PM, PMCs would have to overlap as well.
- Since node groups may contain server machines that are heterogeneous with respect to their hardware, network connectivity, and installed software, PMCs would have to allow heterogeneous servers to be included in them. In consequence, either PMCs would have to be associated with multiple homogeneous resource pools or resource pools would have to be heterogeneous.
- The control logic of the PM would have to be aware of the overlap among PMCs and of the properties of the individual servers. These properties would have to be taken into account when estimating the application performance as a result of changing the PMC membership.
- The optimization logic of the PM would not only have to consider the number of servers that should be added or removed to or from a PMC, but also their type. Similarly, the optimizer must recognize the dependencies between PMCs, e.g., that the decision to remove a server from a PMC is not independent of the decision to remove a server from another PMC, when these PMCs overlap.

As a result, the natural node-group-to-PMCmapping is not possible without significant modifications to the PM model and control logic.

Another approach to concept mapping represents the entire ASM system as a single PMC. In this solution, a PMC has an internal structure that is not visible to the PM, whose only goal is to add and remove server machines to and from the ASM system. Any resource management within PMC, e.g., moving server machines between node groups, is done by the ASM system itself. This approach has significant problems as well.

- Since the ASM may involve heterogeneous servers, the PMC that represents it must allow heterogeneous servers to be included in it. Thus, we

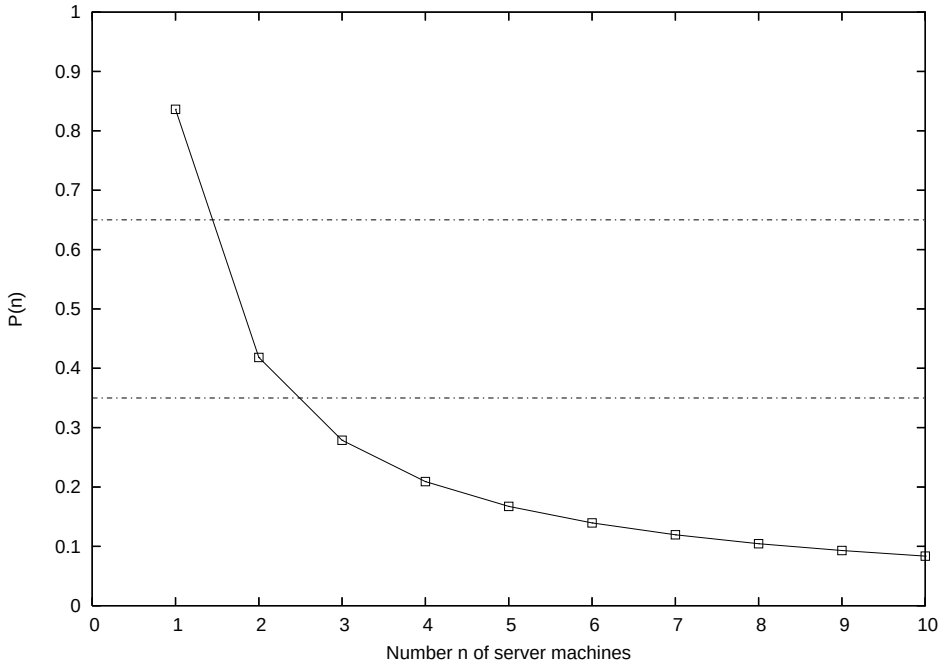

Figure 2. A sample probability of breach curve, showing the target zone.

are faced with the same problem related to modeling and optimizing with respect to heterogeneous resources as it was discussed in the previous approach.

- Moving servers across node group boundaries may require reconfiguring server properties that are outside of the management skills of the ASM environment. For example, it may be necessary to change network connectivity configuration or install or uninstall software libraries. The ASM does not have mechanisms to perform such tasks. We would have to duplicate the functionality of PM within the ASM, instead of integrating with it.

Considering these difficulties, we choose a third approach in which multiple PMCs (in the provisioning manager) map to a single node group (in the ASM). In the event that an ASM node group is made up of heterogeneous servers, servers of different types are placed in different PMCs. Also, if node groups overlap, then PMCs are used to group servers with homogeneous node group membership. This approach has the advantage of allowing us to reuse the modeling constructs of the PM and its functionality. One the other hand, in widely heterogeneous systems, it may lead to a fragmentation of the system model as seen by the PM. Also, due to the lack of a one-to-one mapping between the ASM and PM concepts, the structure of the application environment in the model of PM is obscure, which complicates system configuration and management from a PM perspective.

When configuring the provisioning manager to work with the ASM in a given data center, the system administrator must create PMCs corresponding to the current configuration of the ASM. The provisioning manager requires information to allow it to map its view of the system onto the ASM's view of that same system. Specifically:

- For each PMC, the names of the node groups to which it belongs.
- For each PMC, the name of the ASM instance that controls the corresponding node group (this is required so that the provisioning manager communicates with the correct ASM installation).

This information is provided to the provisioning manager through 'properties' defined (by the system administrator) on the PMC.

4.2. The Objective Analyzer

Recall from Section 3 that the provisioning manager provides an interface by which a given application environment can communicate its need for additional resources, which is expressed in terms of the breach

probability function (PoB). The PoB curve shows, for any possible number of server machines n assigned to the PMC in question, the probability $P(n)$ of violating the SLA of the associated application environment assuming that n server machines are assigned to the PMC.

Whilst the PM comes preinstalled with several Objective Analyzers, none of them currently exploit the information available from the ASM. If the PM is to make correct decisions when it comes to allocating and deallocating nodes to the ASM it is necessary for the PM to have a specific understanding of the ASM environment.

Accordingly, an Objective Analyzer was designed to take advantage of the ASM. This Objective Analyzer communicates with the ASM to obtain information used in its PoB calculation. Observe that there is a discrepancy between the way ASM and the PM express the level of "happiness" resulting from a particular resource allocation. While the PM expresses it in terms of the probability of SLA breach, the ASM uses an objective function. Hence, the Objective Analyzer that allows the ASM to be integrated with the PMC requires two elements: a function that maps server allocation into the value of the objective function of the ASM, and a function that maps the objective function into a PoB.

4.3. Obtaining the utility curve

We have extended the flow controller in the ASM to provide us with an estimated objective function for any given allocation of servers to a particular PMC. To define this concept we extend the definition of the system-wide objective function formulated in Eq. (6). Recall that PMCs are homogeneous. Hence, in discussing the allocation of server machines to PMC, we only need to focus on the number of machines and we can ignore their identity. Let us denote by PMC(k) a cluster of k machines of the type used by PMC. Let us denote by $N\,(\text{PMC}(k))$ the allocation of server machines to the ASM obtained from the allocation currently in effect by placing k server machines in PMC. If k is greater than the number of servers currently in PMC, then the size of PMC is increased. If k is smaller than the number of servers currently in PMC, then the size of the PMC is decreased. Otherwise, the configuration of the ASM is unchanged.

We define the utility U of allocating n server machines to a PMC as follows.

$$U(n) = \min_{p\in P}\min_{d\in D}\min_{c\in C} U_c(\bar{P}_{p,d,c}(N\,(\text{PMC}(n)))) \quad (12)$$

$\bar{P}_{p,d,c}(N\,(\text{PMC}(n)))$ is the expected performance result for the *flow* identified by ODR p, DC d, and service class c, which is obtained as described in Section 2.2 in the context of computing input for the placement controller. While performing the calculation we take into account the hypothetical allocation of resources expressed by $N\,(\text{PMC}(n)))$ rather than the current allocation. In particular, the set of constraints in Eq. (10) covers all and only nodes in $N\,(\text{PMC}(n)))$ rather than nodes in the current configuration.

4.4. Mapping utility into probability of breach

The mapping of utility values for a set of possible server machine allocations to a corresponding set of PoB values is achieved in two steps. The first step aims to fit a negative exponential function through the set of available utility values using standard regression techniques. This step is necessary since the ASM may provide the estimated utility values for only a small subset of all possible server machine allocations (for reasons of computational costs). The second step maps the fitted exponential function to a sigmoid PoB function for all possible allocations. This section provides further details of these two steps.

Currently, the ASM provides the set of utility values $\mathbf{U}(n) = \{U(n-2), U(n-1), U(n), U(n+1), U(n+2)\}$ to the Objective Analyzer where n is the current allocation of server machines. The functional form of the class of utility functions is based on the objective function detailed in Eq. (7) which specifies that the utility values are bounded from above by 1.0, and the utility value is zero when the performance result metric is equal to the associated threshold value. There is no lower bound on the utility value. We fit the utility data $\mathbf{U}(n)$ to a negative exponential function of the functional form $\hat{\mathbf{U}}(x) = 1 - e^{a-bx}$ using standard weighted linear least squares method, where x is the level of allocation, and the weight associated with each data point is determined by the relative difference in utility with its adjacent points.

To convert the $\hat{\mathbf{U}}(x)$ function to the PoB curve, we map x_1, the root of the equation $\hat{\mathbf{U}}(x) = 0$ to the upper threshold of the target zone for the PM where PoB $P(x_1) = p_1$. At this level of allocation, the performance result metric begins to fail in meeting the associated threshold. On the other hand, positive utility values signify that the ASM is meetings its goals, and hence the first integral value of x for which $\hat{\mathbf{U}}(x) > 0$ is mapped into the target zone. This is accomplished by assigning x_2, the mid-point of the first two integral values of x for which $\hat{\mathbf{U}}(x) > 0$, to the lower thresold of the target zone for the PM where PoB $P(x_2) = p_2$.

Given these two points on the PoB curve (x_1, p_1) and (x_2, p_2), where x_i is the allocation level with PoB $P(x_i) = p_i$, we fit a sigmoid function $P(x) = 1/e^{-p+qx}$ to obtain the probability of breach for any level of allo-

cation x.

It is important to note two issues regarding the mapping scheme detailed above. First, the mapping from the utility values from the ASM to the probability of breach value for the PM is necessary because the two measures are incommensurate. If both ASM and the PM used the same performance measure, the mapping would be considerably simpler. Indeed, we have performed preliminary studies where both ASM and PM use similar probability of breach measure to communicate through the Objective Analyzer thereby making the mapping process very straightforward.

On the other hand, it is likely that interacting autonomic systems will often use incommensurate metrics to measure performance in which case some form of mapping will be necessary for the entities to interact harmoniously. While we have detailed one approach above, we do note that machine learning approaches could be gainfully employed in such mappings to make the interactions between autonomic systems more cohesive. For example, insted of fitting a new $\hat{\mathbf{U}}(x)$ to each new set of data $U(n)$, a learning system could include historical utility data in providing consistent utility estimates that is less susceptible to biases and random fluctuations in $U(n)$.

4.5. Damping in the Objective Analyzer

The Objective Analyzer must also take account of the reality that, because ASM is a learning system that must adapt to the addition and removal of server machines, the utility curves that the Objective Analyzer receives from ASM will fluctuate. This fluctuation is particularly significant in the immediate aftermath of an allocation change — for example, in the case of the addition of a new node, ASM will require time to use the new node and to adapt to its characteristics. In this time, it is likely that the utilities will fluctuate sufficiently to push the $P(n)$ outside the target zone, and trigger another allocation decision. Making an allocation change in the period immediately following an earlier allocation change will only compound the fluctuations, and increase the likelihood of yet a third allocation change, and so on.

Consequently, the Objective Analyzer contains damping code that will prevent allocation decisions for a particular PMC from being made in the period immediately following the number of server machines in thatPMC changing. Fortunately, the provisioning manager permits an Objective Analyzer to return a 'default $P(n)$ surface' — this is a surface for which, if n is the currently allocated number of nodes, $P(n) = 0.5$; for any n less than the current allocation, $P(n) = 1.0$; and, for any n greater than the current allocation, $P(n) = 0.0$. When the provisioning manager receives this 'default $P(n)$ surface' from an Objective Analyzer, it will not (in the absence of resource contention) make any allocation changes to the PMC in question. If there is resource contention (that is to say, if other PMCs in the system are starved of resources), then allocation changes may still be made — it is not possible (nor should it be) for an Objective Analyzer to expressly prevent the provisioning manager from making allocation changes to its PMC.

The time period for which this default $P(n)$ surface is provided by the Objective Analyzer to the provisioning manager can be overridden on a per-PMCbasis.

5. Experiments and results

In order to demonstrate the integration of the provisioning manager and the ASM, experiments were performed in which an installation of the provisioning manager was configured to manage an ASM installation, and load was injected into the ASM installation.

5.1. Overall experimental setup

For reasons of clarity, both experiments described herein are quite simple — more complicated experiments are possible, but describing the interactions in such systems would only be intrusive. Both experiments used a single ASM installation with four server machines. The server machines were homogenous — they were IBM eServer xSeries 335 machines, each containing a single Intel Xeon 3.2GHz processor (with 512KB of Level 2, and 2MB of Level 3, cache) configured with hyperthreading enabled (so that each physical processor presents as two virtual processors). Each machine had 3GB of RAM.

The goals used in the ASM configuration were the same throughout — 400ms average response time for all requests. The properties of the test application and the server machines are such that a single server machine can serve between 25 and 30 clients and still meet the 400ms average response time target. The demand, measured in number of clients sending requests to the ASM, is simulated using a single closed-loop load generator with an adjustable number of clients.

To make the operation of the experiments easier, the PM was placed into 'semi-automatic' mode. In this mode, the PM does not act immediately on its decisions, but instead asks, via its UI, for approval from an administrator. This introduces delays at some stages in the process when the PM waits for the administrator to accept a recommendation, but also permits greater control

over the experiment.

In order to accelerate some aspects of the testing, the damping period (see Section 4.5) was set to two minutes for all Clusters.

5.2. Single Node Group

In the first experiment, the ASM system was configured with a single Node Group, containing (initially) two server machines. A single DC was created, backed by this Node Group, and a single application (a simple CPU consuming test application) was installed into that DC.

Readers familiar with [3] will recognise that this experiment uses the same configuration as the experiment discussed in that paper[1].

Figure 3 shows some characteristics of the test system over time. The horizontal axis shows seconds elapsed since the start of the test run. The graphs show (from the bottom up):

- The client count, simulated by a single load driver,
- The response time, as perceived by the load driver — response times are aggregated over 15 second intervals — and the desired response time (the horizontal dotted line),
- The value of $U(n)$, where $U(n)$ is the utility of the current server machine allocation (see Section 4.3); and
- The value of $P(n)$, where $P(n)$ is the Probability of Breach for the Node Group with the current server machine allocation (see Section 4.4). The horizontal dotted lines on this topmost graph show the target zone, as in Figure 2.

At time 0, the load is a constant 45 clients. At around 120s elapsed time, the client count is increased to 50 clients — the response time curve immediately trends slightly upwards, $U(n)$ curve trends slightly down, and $P(n)$ curve trends slightly up. At around 400s elapsed time, the load is increased again to 55 clients — again, the response time curve goes up, $U(n)$ curve goes down, and $P(n)$ goes up. However, the two server machines can still handle the load, so $U(n)$ remains above 0 and $P(n)$ remains below the upper threshold of the target zone.

At around 750s elapsed time, the client count is increased again to 60 clients. The response time curve rapidly rises over the 400ms response time target, pushing $U(n)$ below 0 and $P(n)$ above the upper threshold of the target zone. At 790s elapsed time, the PM recommends the addition of a single server machine to the Node Group, and at 895s elapsed time, that recommendation is accepted. Between these two times, $U(n)$ continues to trend downwards as the response time settles towards its new, higher, level — and $P(n)$ continues upwards. At 895s elapsed time, the allocation begins, and the allocation process continues until 1157s elapsed time.

After 1157s elapsed time, the response time immediately drops (back to less than the target of 400ms), as the new server machine starts to serve requests. The damping period discussed in Section 4.5 begins as soon as the count of server machines changes — that is to say, once the allocation has completed. Consequently, whilst the allocation is in progress, and then during the damping period, no values for $U(n)$ or $P(n)$ are gathered [2]. At the end of the damping period, data gathering resumes, and $U(n)$ (recall that n is now 3, reflecting the new allocation) has returned to above 0. Similarly, $P(n)$ is now back within the target zone.

5.3. Two Node Groups

For the second experiment, the ASM system was configured with two Node Groups. Each of these Node Groups is identical to the single Node Group used in the previous experiment, and are here referred to as A and B. These Node Groups do not overlap (that is to say, they do not share any server machines). Note that, in this experiment, the PM has no server machines available in the resource pool for allocation to the ASM system — in order to increase the allocation of one Node Group, it will first be necessary to reduce the allocation to the other Node Group.

Figure 4 is comparable to Figure 3 for the first experiment — however, in this case, two $P(n)$ and response time curves are shown, one for Node Group A and one for Node Group B.

In a similar manner to Figure 3, $P_A(n)$ increases when the number of clients directing their workload at A is raised (at around 245s elapsed time). $P_B(n)$ for B is not affected by this change, as the workload to NG B has not changed. However. as $P_A(n)$ has left the target zone, the PM must consider reallocation options. As there are no server machines in the resource pool, the PM might at first appear to have no options — this is,

[1] However, readers are cautioned not to compare the response time figures of the test application here and the test application in [3] — we are here demonstrating the linkage between the ASM and the PM, and have not attempted to optimize the ASM installation for speed.

[2] Because, as discussed in Section 3, the PM runs on a 30 second decision cycle, the next values for $U(n)$ and $P(n)$ are gathered on the first decision cycle after the damping period ends — so, in these experiments, the effective damping period could be as long as 150s.

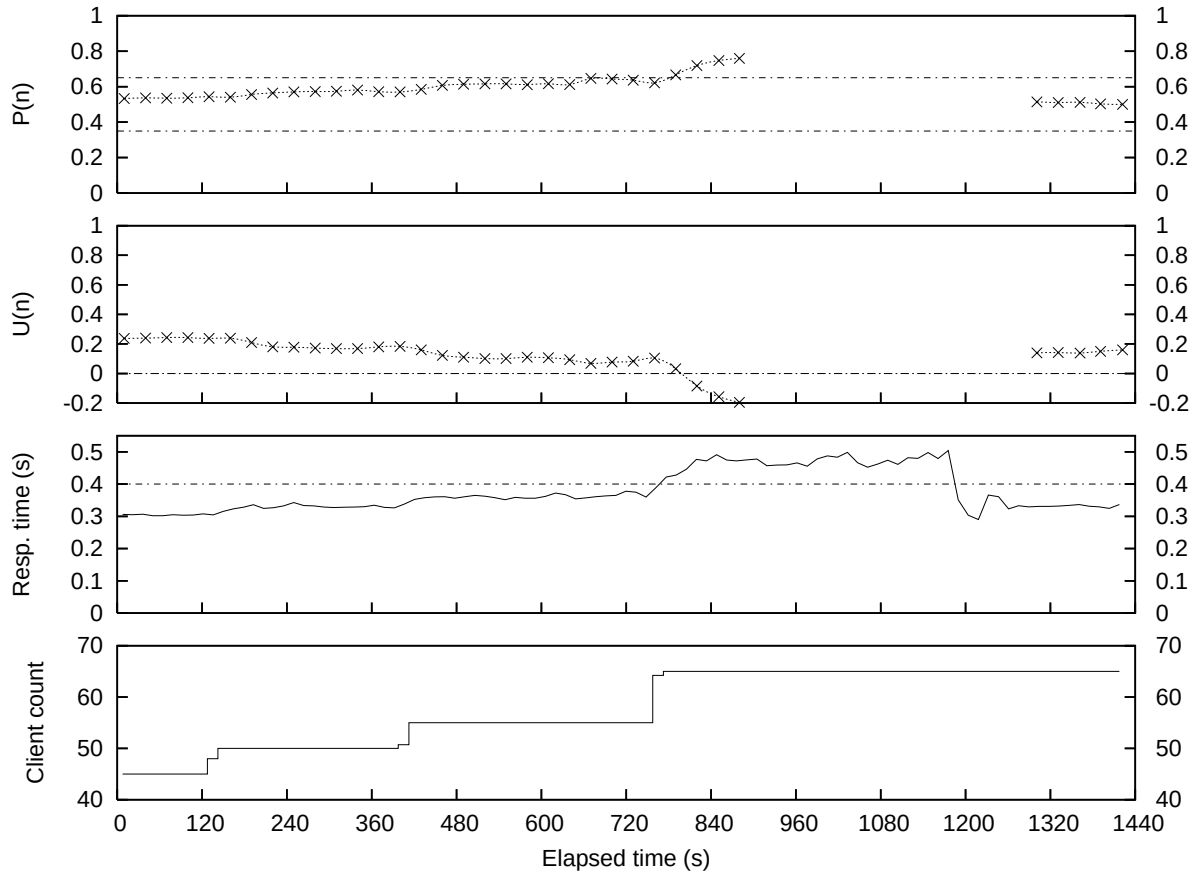

Figure 3. Single Node Group experiment

however, not the case. As the workload the B is small, not only is $P_B(2)$ inside the target zone, but $P_B(1)$ is also (just) inside the zone. Also, $P_A(3)$ is inside the target zone — that is to say, if the PM moves one of the server machines from B to A, both A and B will be meeting their goals.

In addition, the overall breach probability of the system is decreased if this reallocation is performed, since:

$$P_A(3) + P_B(1) < P_A(2) + P_B(2) \qquad (13)$$

Consequently, at 345s elapsed time, the PM recommends the removal of a server machine from B. This recommendation is accepted at 350s elapsed time, and the removal runs until 571s elapsed time. At 582s elapsed time (the provisioning cycle after the removal has finished), the PM recommends the addition of the now-spare server machine to A — note that the damping period does not prevent this recommendation, as it does not relate to the PMC for which a change was recently made. That is to say, B is in the damping period, but A is not, and so recommendations can be made for A. This recommendation (to add a server machine to A) is accepted at 587s elapsed time, and runs until 858s elapsed time.

Two minutes after the remove operation completes, the Objective Analyzer begins to compute PoBs for B once again – these points can be seen starting at 702s elapsed time. Similarly, two minutes after the add operation completes, the Objective Analyzer begins to compute PoBs for A once again – these points can be seen starting at 1005s elapsed time. Recall that when the PoBs reappear on the graph, they are the PoB of the current (that is to say, the new) allocations.

The end of the graph shows the system performing within the goals — the $P_B(n)$ is (just) within the target zone, and $P_A(n)$ is well within it. And the response times show that both applications are meeting their targets.

5.4. Single Node Group with varying load

This experiment used a configuration similar to the simple single Node Group experiment discussed in Sec-

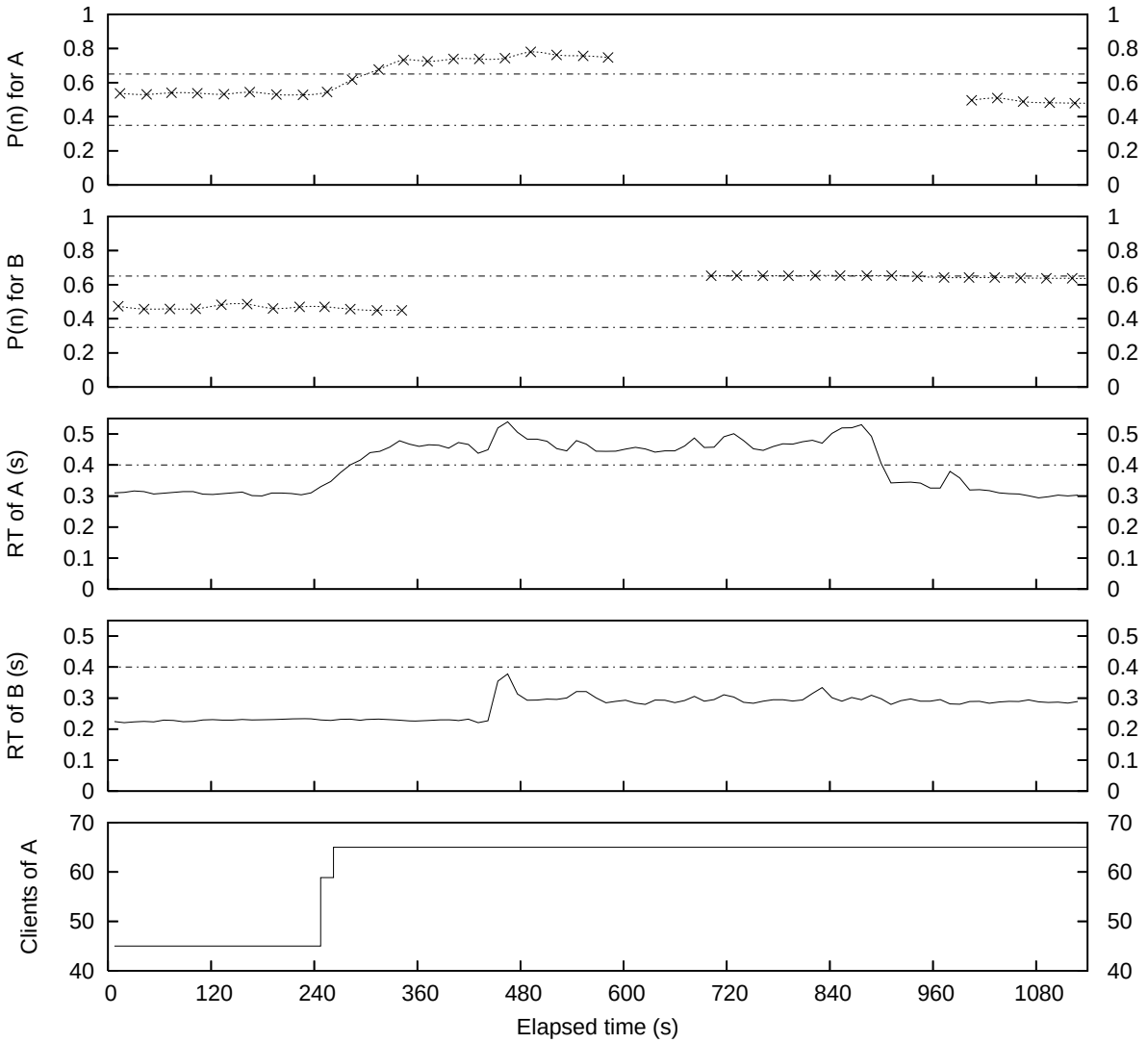

Figure 4. Two Node Group experiment

tion 5.2 — however, in this case the Node Group initially contained only one server machine. The other three server machines were available in the PM's resource pool for allocation when needed. For this experiment, the PM was placed into 'automatic' mode, wherein allocation decisions are acted on automatically and immediately without intervention from a administrator.

In order to provide a varying load over time, we configured the load generator to produce a realistic emulation of stochastic bursty time-varying demand by employing a time series model of Web traffic developed by Squillante et al. [9]. The number of clients in the closed-loop load generator is reset every minute according to this model with hard lower and upper thresholds of 15 and 100 clients respectively.

The system was left to run overnight, and the results collected.

Figure 5 shows the number of clients and the desired server machine allocation over time. The 'desired server machine allocation' reflects, for each provisioning cycle of the PM, how many server machines the PM decides that the PMC needs. If this desired allocation number is different from the current allocation number, then a reallocation will be performed — but this reallocation takes several minutes (as was shown in earlier experiments). The current server allocation precisely matches the desired server allocation with a time lag of approximately five minutes. The figure shows that, in spite of this delay, the ASM-PM interaction is able to successfully allocate server machines under widely varying load conditions.

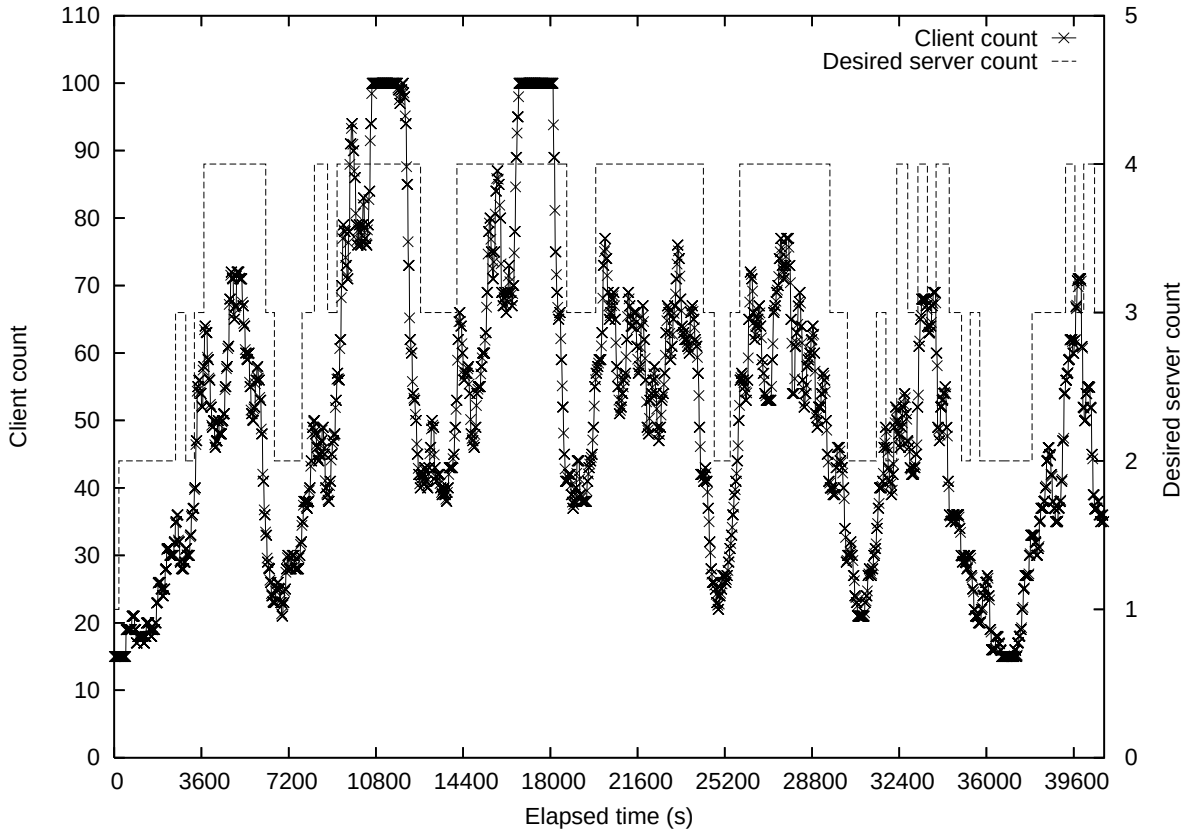

Figure 5. Single Node Group with varying load: client count and desired server allocation over time

Figure 6 shows the overall allocation changes that emerge from the ASM-PM interactions. Using aggregate data, the figure plots the likelihood of the PM determining that the desired server allocation is one, two, three, or four for various instantaneous client counts. The client counts are grouped into bins of size 10. As expected, the probability of the PM wanting a larger server machine allocation increases as the client count increases.

This experiment demonstrates that the algorithms presented earlier in this paper enable the system to manage resources according to performance goals both in allocating transactions within the currently assigned set of server machines, and in changing that assignment in response to varying load.

6. Architectural analysis

In this section, we will analyze the system described above in terms of the architecture for self-managing systems presented in [2], and outline how some emerging standards for distributed manageability apply to this example of self-management. We begin with an overview of the architecture and its essential concepts.

6.1. Autonomic architecture

The basic component in the architecture is the "self-managing resource". A self-managing resource is a component that is responsible for managing its own behavior in accordance with policies, and for interacting with other self-managing resources to provide or consume IT services.

In this approach, every component of an autonomic system is a self-managing resource. This includes basic computing resources such as databases, storage systems, or servers; higher-level components with some management authority, such as workload managers or provisioners; and resources that assist others in performing their functions, such as sentinels, brokers, or service registries.

All self-managing resources have certain characteristics.

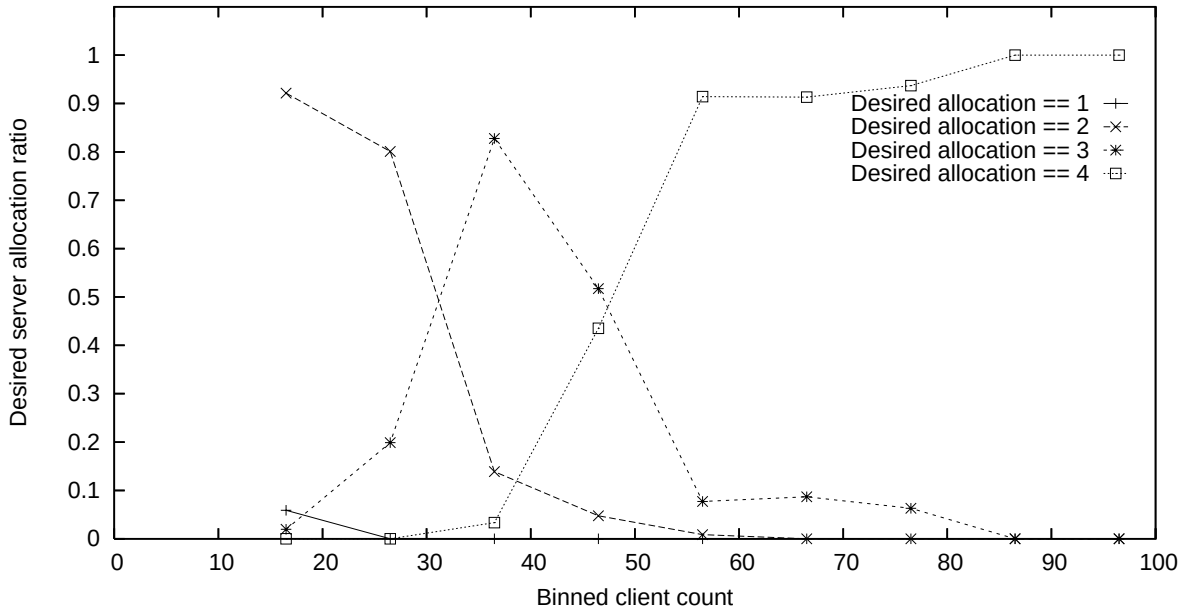

Figure 6. Single Node Group with varying load: relationship of client count to desired server allocations

- First, self-managing resources are in fact self-managing; that is, each one is responsible for configuring itself internally, for coping with internal failures, for optimizing its own behavior, and for protecting itself from external attack. A self-managing resource handles problems locally whenever possible, including problems caused by the failure of other entities upon which it depends.

- Second, self-managing resources are capable of forming and abiding by agreements with other self-managing resources. Agreements are the means by which multiple self-managing resources are composed into larger autonomic systems. In order to allow the formation of agreements, each self-managing resource must describe itself accurately, and in such a way that the description is accessible to and understandable by other self-managing resources. And each self-managing resource must be capable of negotiating (if only trivially) to establish agreements.

- Third, self-managing resources manage their own behavior and interactions so as to meet their obligations, by appropriately adjusting their own internal parameters and by drawing upon the other resources in the system. There are two types of obligations to which a self-managing resource may be subject: agreements, and policies. A self-managing resource must not accept any service request which would violate its policies or agreements, and it must not accept any proposed agreement or policy that would cause a violation of its existing agreements or policies. It must have sufficient analytic capabilities to support these functions.

As used in the architecture, policies are descriptions of constraints and requirements upon the behavior of one or more self-managing resources, or more generally (as described in [10]) any description of the value of one or more possible states of the system. Policies can range from very simple declarative rules about the low-level behavior of the system to high-level business objectives [11], and utility functions [12]. To allow policies to be communicated between self-managing resources, they must be represented in a standard external form.

A significant design pattern enabled by the autonomic architecture is **goal-driven self-assembly** (described in detail in [13] and [14]). This pattern exploits the ability of self-managing resources to describe themselves, locate each other, and form agreements. In the pattern, self-managing resources determine at runtime what services they will need to obtain in order to carry out the policies that apply to them, and dynamically locate other resources capable of supplying those services, by consulting a service registry in which all

available self-managing resources announce themselves and their capabilities. (This service registry is similar in concept to those used in multi-agent systems [15] and dynamic e-business systems [16].)

6.2. Analysis of the current system

Since the concrete system described in this paper is built on existing systems management software and the autonomic architecture that we outline is relatively new, the system is not completely conformant to the architecture. But since the system has significant autonomic features, and has the advantage of actually existing, we can use it to illuminate what a full implementation of the architecture might look like, and what practical requirements will apply to it.

The first task in analyzing the current system against the autonomic architecture is to decide where to draw the lines — how to divide the system into interacting self-managing resources.

At least two possibilities present themselves here: we could draw the line between the objective analyzer and the ASM, or between the objective analyzer and the provisioner. In the former case the key data items communicated between the managers are the expected utilities associated with possible resource allocations, as described in Sections 2 and 3. In the latter case, it is the Probability of Breach surface.

Determining how to divide an existing (or proposed) system into self-managing resources is similar in many ways to determining how to divide a program into modules, or more generally how to create an information model for any real or IT system. To some extent, all of these processes are matters of art as much as they are matters of science. But there are general principles that apply; the division should maximize the reusability of the components (self-managing resources, program modules, object classes) that result, and the cleanliness of the interfaces between them.

In the case of the system we described in [3], it was most natural to draw the line between the the objective analyzer and the ASM. We considered the ASM, including the controllers and the objective analyzer described above, to be one self-managing resource, and the provisioning manager to be another. This was preferable to the alternative because the information passed between the ASM and the objective analyzer in that system was relatively specific to the ASM and not as generally useful as the probability of breach data.

In the current system, however, both probability of breach data and utility estimates are conceivably useful to other components; it is easy to imagine the ASM usefully providing either one to other self-managing resources, such as other provisioning managers, problem determination systems, monitoring consoles, and so on. Our current belief is that utility estimates are somewhat more general than probability of breach curves, but further experience is required to validate this. We will explore both the possibility of the ASM reasoning directly in terms of probability of breach, and of the PM reasoning in terms of probability.

Similar principles apply when we consider a related question: whether the ASM itself should be divided into multiple self-managing resources. Since the flow manager and the placement controller are relatively disjoint, and the signals that pass between them well-defined, it would be possible to separate them into separate resources that would then be rejoined through agreements and published interfaces. At present, however, it does not appear that the resulting interfaces would be particularly reusable; if nothing but the placement controller would ever want to collaborate with the flow manager, and vice-versa, they are best kept within a single resource.

On the other hand, if the server machines themselves were self-managing resources with well-defined service interfaces, some advantages would accrue. Currently the methods by which the ASM and the PM communicate with and cause actions on the server machines are somewhat *ad hoc* (shell scripts run via ssh, for instance). We are now constructing a prototype system in which server machines are explicitly represented as self-managing resources, with which the management resources interact in well-defined ways.

In the current system, the relationship between the ASM and the provisioning manager is conceptually simple: the former provides the latter with estimates of how it would behave at various levels of resource, and the latter supplies resources to the former in a way that increases the overall performance of the system. Making this relationship explicit, by representing it as an agreement expressed and documented in a standard agreement language, would allow other components of the system, such as visualization tools and problem-determination services, to access and reason about the relationship. It would also allow both of the managers to explicitly represent, in a common format, the set of servers that they are collaboratively managing; in the current system this set is maintained separately in each manager, which involves some duplication of effort and information.

Having the two managers explicitly construct their relationship via agreement formation would also reduce the amount of manual configuration required, since the managers could exchange the necessary information in a principled way as part of the initial negotiation.

Similarly, we can regard the service class definitions and threshold and importance values that serve as the inputs to the application server's objective-function calculation as policies, or service-level agreements, that partially constrain the behavior of the ASM. By setting these values, the administrator controls the goals to which the autonomic system will manage itself. These values are currently entered into the system through the administrative user interface. Exposing these values explicitly, in a standard policy or SLA language, would add to the system's flexibility, and allow it to acquire these policies from other policy-deployment systems that are currently being designed, and enable those policies to be computed automatically from higher-level business goals and thence deployed into the system.

6.3. Standards for manageability

Given any pair, or any small set, of software components, it would be possible to design *ad hoc* mechanisms, languages, and protocols to acheive the intercomponment aims of the architecture presented above. Once the hard work of modelling server machine behavior, forecasting demand, and computing utility estimates and probability of breach surfaces has been done, the task of communicating the results from one given component to another is comparatively trivial.

In the larger world, though, this kind of pairwise and setwise design does not scale well; designing and writing new code for each new pair of components that need to interact would be prohibitively inefficient. Standards are therefore needed for communicating the common kinds of information, and carrying out the common kinds of interactions, that this architecture and other related architectures require. This task is no longer trivial; the development of effective standards requires solving significant technical, as well as organizational and political, problems.

We will not attempt in this paper to show in detail how the system we describe here could make use of the emerging standards for distributed system manageability, but having described the system and its interactions, it seems worthwhile to mention the most relevant standards, and suggest how they apply to the problems encountered in creating a scalable architecture for such system.

The manageability standards that we describe here are built around the Web Services architecture described in [17]. The Web Services architecture provides a general means for software entities to interact in a distributed environment; it includes a general scheme (the Web Services Description, expressed in the Web Services Description Language WSDL) for entities to describe the services that they provide in terms of their names and the names and types of their parameters.

The Web Services architecture provides the basic interoperability standard. Further work is building on this base to provide standard ways to represent and describe stateful resources, their capabilities, and their relationships, in ways that meet many of the requirements of the autonomic computing architecture.

The Web Services Resource Framework [18] provides a set of standards for modeling stateful resources using web services, including standard ways of representing and accessing the properties and lifetimes of resources and groups of resources, and of subscribing to and receiving topic-based notifications about resource-related events.

Higher-level standards build in turn upon this resource framework. The Management Using Web Services (MUWS) specifications [19] [20] apply specifically to manageability and management, specifying how resources expose their identities, manageability characteristics and capabilities, management-relevant metrics, operational state, and relationships to other entities in the system. The WS-Agreement specification [21] specifies languages and protocols that enable one component to announce that it is willing to consider entering into various sorts of relationships, and another component to propose actually doing so.

While many of these specifications and standards are still in active development, together they offer a promising suite of common methods for satisfying the requirements of the autonomic architecture.

In terms of the system described in this paper and the analysis presented in the previous section, these standards would allow us to model the provisioning manager and the node group manager as web services, to take advantage of off-the-shelf development and runtime libraries for creating web services. Building on the Web Services Resource Framework, the MUWS specification shows us how the node group manager could expose in a standard way the fact that it is capable of producing probability of breach data, and the Web Services Description Language shows us how to describe the names and parameters of the calls that the provisioning manager can make to request and obtain that data. The MUWS and WS-Agreement specifications offer standard ways of representing the relationships between the components in a way that other parts of the system can understand, and also allow each component to advertise the kinds of relationships into which it can enter.

All of these specifications working together take us some of the way toward realizing the architectural goals described above in systems like the one this paper de-

scribes, in a way that avoids the scaling problems of pairwise solutions.

They do not take us all the way; not only are the specifications not yet complete, but there are still significant open problems in resource modeling and specific semantics. The MUWS standard for manageability characteristics tells us how the node group manager can announce its capabilities, but it does not specify a particular capability name corresponding to the production of probability of breach data or utility estimates. Much more work remains to be done in bringing these emerging standards to bear on practical problems; we urge other researchers to take these standards seriously, and validate them against the problems that they face.

7. Future Work

As well as increasing the transparency and interoperability of the system using the emerging manageability standards discussed in the previous section, we plan to study a number of other extensions and refinements of the present system.

Our experiments so far have concentrated on objective functions based on average response time, as described in section 2 above. It would be interesting to explore the dynamics of the system under the other type of objective function supported by the ASM: percent of response times above a given threshold. While we believe that the utility calculations and basic interaction patterns will continue to apply, experimental confirmation is highly desirable.

As mentioned in section 4 above, the computation of and use of utilities in the system would be simplified if the ASM and the PM used the same, or more simply commensurable, performance measures. We are working on more general measures of performance and utility that could be used to satisfy the computational requirements of both components. We are also investigating machine-learning approaches to increase the quality of the mapping with experience when measures remain incommensurable.

Another interesting extension to the system would involve using the ASM and the PM to manage the on-demand routers (ODRs) as well as the application servers. Like application servers, ODRs can be started and stopped on the available server computers; unlike application servers, there is no explicit performance goal associated with the ODRs. It would be relatively straightforward to create an Objective Analyzer that would perform appropriate utility calculations and allow the PM to add and remove ODRs in respones to load and performance, although determining the proper effective performance goal for the "cluster" of ODRs will require invention.

In the longer term, it will be necessary to derive the performance goals for individual applications from business goals expressed in higher level terms. Performing this type of derivation is one of the major challenges in the creation of self-managing systems.

8. Conclusions

We have described an autonomic computing system in which two managers collaborate to optimize the behavior of the infrastructure against human-specified goals. The managers themselves have autonomic features, and the information that they exchange allows them to perform optimizations that neither would be capable of alone. We have described how the application server routes and classifies requests and estimates the impact of having more, or fewer, server nodes available for servicing those requests. We have also described how the application server's impact estimates are used by the provisioning manager to do higher-level orchestration within the data center, and we presented the results of experiments validating the approach.

As autonomic systems management technologies such as these become more sophisticated and more widespread, there will be an increasingly urgent need for a unifying architecture and common standards that allow them to interoperate, both to avoid conflicts and to enable synergies. We have analyzed the current system in terms of one proposed architecture for autonomic systems, based on the concept of self-managing resources, and a set of emerging Web Services manageability standards, and we have outlined some areas for future research work.

9. Acknowledgements

We would like to thank U. Gopalakrishnan for providing us with the necessary modifications to the load generator to produce time-varying demand as detailed in Section 5.4.

References

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–52, 2003.

[2] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An architectural approach to autonomic computing," in *First International Conference on Autonomic Computing*, 2004.

[3] D. M. Chess, G. Pacifici, M. Spreitzer, M. Steinder, A. Tantawi, and I. Whalley, "Experience with collabo-

rating managers: Node group manager and provisioning manager," in *Second International Conference on Autonomic Computing*, 2005.

[4] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic application placement under service and memory constraints," in *4th International Workshop on Efficient and Experimental Algorithms*, Santorini Island, Greece, May 2005.

[5] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for cluster based web services," in *8th IFIP/IEEE International Symposium on Integrated Network Management (IM 2003)*, 2003.

[6] IBM Corp., "IBM Tivoli Intelligent Orchestrator — Product overview," http://www.ibm.com/software/tivoli/products/intell-orch/.

[7] E. Manoel, S. C. Brumfield, K. Converse, M. DuMont, L. Hand, G. Lilly, M. Moeller, A. Nemati, and A. Waisanen, "Provisioning On Demand: Introducing IBM Tivoli Intelligent Orchestrator," Dec. 2003.

[8] F. J. De Gilio, "Orchestriating grid workloads — neither feast nor famine," http://www-128.ibm.com/developerworks/grid/library/gr-feast/, 2004.

[9] M. S. Squillante, D. D. Yao, and L. Zhang, "Internet traffic: Periodicity, tail behavior and performance implications," in *System Performance Evaluation: Methodologies and Applications*, 1999.

[10] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, 2004.

[11] S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, and S. Wasserkrug, "Autonomic self-optimization according to business objectives," in *First International Conference on Autonomic Computing*, 2004.

[12] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic computing," in *First International Conference on Autonomic Computing*, 2004.

[13] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: Experiences with a prototype autonomic computing system," in *First International Conference on Autonomic Computing*, 2004.

[14] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "A multi-agent systems approach to autonomic computing." in *AAMAS*, 2004, pp. 464–471.

[15] K. Sycara, "Multi-agent infrastructure, agent discovery, middle agents for web services and interoperation," in *Mutli-agents systems and applications*, J. G. Carbonell and J. Siekmann, Eds. Springer-Verlag New York, Inc., 2001.

[16] "Introduction to UDDI: Important features and functional concepts," http://uddi.org/pubs/uddi-tech-wp.pdf, 2004.

[17] W3C Web Services Architecture Working Group, "Web services architecture," http://www.w3.org/TR/ws-arch/, 2004.

[18] Globus Alliance, "The WS-Resource Framework," http://www.globus.org/wsrf/, 2004.

[19] "Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1," http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part1-1.0.pdf, 2004.

[20] "Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 2," http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part2-1.0.pdf, 2004.

[21] "Web Services Agreement Specification," https://forge.gridforum.org/projects/graap-wg/document/WS-AgreementSpecification, 2005.