Enabling Policy-Driven Self-Management for Enterprise-Scale Systems

Vibhore Kumar, Brian F. Cooper†, Greg Eisenhauer, Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{vibhore, eisen, schwan}@cc.gatech.edu

Yahoo! Research†
2821 Mission College Blvd.
Santa Clara, CA 95054
cooperb@yahoo-inc.com

Abstract

It is becoming increasingly evident that the growing complexity of the enterprise-scale IT systems, both in terms of underlying infrastructure and the user expectations, is quickly surpassing the abilities of system administrators to efficiently manage these systems. Existing solutions for dealing with this management problem have not been up to the mark on several fronts which include the lack of support for automated reasoning and learning, the failure to scale for large systems and their inadequate support for including the administrator in the self-management mechanisms, etc. In this paper, we describe a policy-driven self-management solution for such enterprise-scale systems with a focus on the issues of (1) wisdom incorporation - *the solution allows system administrators to embed their knowledge into the mechanisms used for enabling self-management, (2)* scalability - *our solution uses a novel system state-space partitioning scheme to handle a multitude possible system events and the corresponding corrective actions, (3)* dynamism - *our solution is able to make suitable models of the system behavior for various system state-space partitions and this allows for abilities like policy-adaptation, policy learning and conflict resolution, and finally (4)* trust - *our solution is able to expose its reasoning for adaptations to the administrator and provides interfaces using which the 'degree' of self-management can be fine-tuned. Finally, we present results from simulation experiments which corroborate the claims made in this paper.*

1 Introduction

We consider large systems that form an integral part of an enterprise's IT infrastructure. Examples of such systems include the one supporting the enterprise's website, or the inventory management subsystem, or even the distributed system supporting the operational information system. The administrators managing these systems are often expected to make the system constantly meet some real-time processing constraints, offer differentiated QoS, offer high-availability and a list of other desirable features. However, the fact is that even the occurrence of seemingly normal and regular events like load changes, node and link failures, software patches or a modification of some environment parameter can cause the system to behave in unexpected ways and not meet its objective, leaving even the experienced system administrators searching for solutions. This inability of administrators to efficiently manage such systems can be attributed to growing complexity and the dynamic conditions under which such systems operate, making it impossible for human administrators to keep track of all system variables and the ways in such system variables may interact with each other. As the complexity of such systems keeps growing and as the enterprises keep becoming more and more reliant on them it is becoming evident that it is beyond the scope of human administrators to efficiently manage these systems. This motivates the need for a solution that can enable self-management for such systems driven by high-level business goals.

The state-of-the art in enabling self-management for enterprise-scale systems includes the use of monitoring tools like IBM's Tivoli, HP's OpenView and other related products. These tools are equipped with methods for system monitoring and graphically displaying system status to administrators. However, their functionality is still rudimentary when it comes to automated symptom determination or reasoning and taking appropriate corrective actions, which in part can be attributed to the general nature of these tools. Researchers in the general domain of self-management have proposed methods for learning the system behavior, while others have suggested exhaustive enumeration of policies - both of which fall short for large systems as the learning methods can be exceedingly slow and exhaustive enumeration of policies may not be possible. Some of the work done for enabling self-management in such systems requires the use of utility-functions that define a relationship between a subset of system variables and the system optimality. How-

ever, based on our experience with using such functions, it may be very hard to find a function that represents the optimality of a system configuration for large systems (e.g. an airline operational information system).

We are developing a framework for enabling self-management of enterprise-scale systems such that the system can then 'autonomically' manage itself in accordance with the goals specified at a high-level. Our solution makes use of existing 'wisdom' and high-level goals in conjunction with learning techniques to dynamically refine and learn policies that are then used for managing the system. We represent the system state at any time as a point in a multi-dimensional space (we refer to it as the system state-space), where each axis represents an identifiable system variable (e.g. end-to-end delay, throughput, etc.). The techniques employed by our self-management algorithm make use of a novel state-space partitioning scheme to achieve scalability and to efficiently learn and improve policies. The following are the main contributions of our framework for enabling policy-driven self-management of enterprise-scale systems.

- *Wisdom Incorporation* - We believe that self-managing systems can benefit from the common wisdom that exists for that application domain. Our techniques for policy-driven self-management are such that the system administrator can easily embed his knowledge about the system into the mechanism used for enabling self-management.

- *Scalability* - In an enterprise-scale system there is a large set of variables that constitute the system's state-space and these variables are not independent of each other. In most cases it might not be possible to determine the interactions between the large number of variables and consequently build a model of the system. However, by making use of a novel state space partitioning scheme we are able to build micro-models for system sub-spaces.

- *Dynamism* - It is important that the system should be able to modify the existing policies, learn new policies or even discard older policies when the operating conditions or the environment changes (e.g. new nodes being added to the middleware). The ability of our system to incorporate new observations into the existing system model helps us to deal with this problem

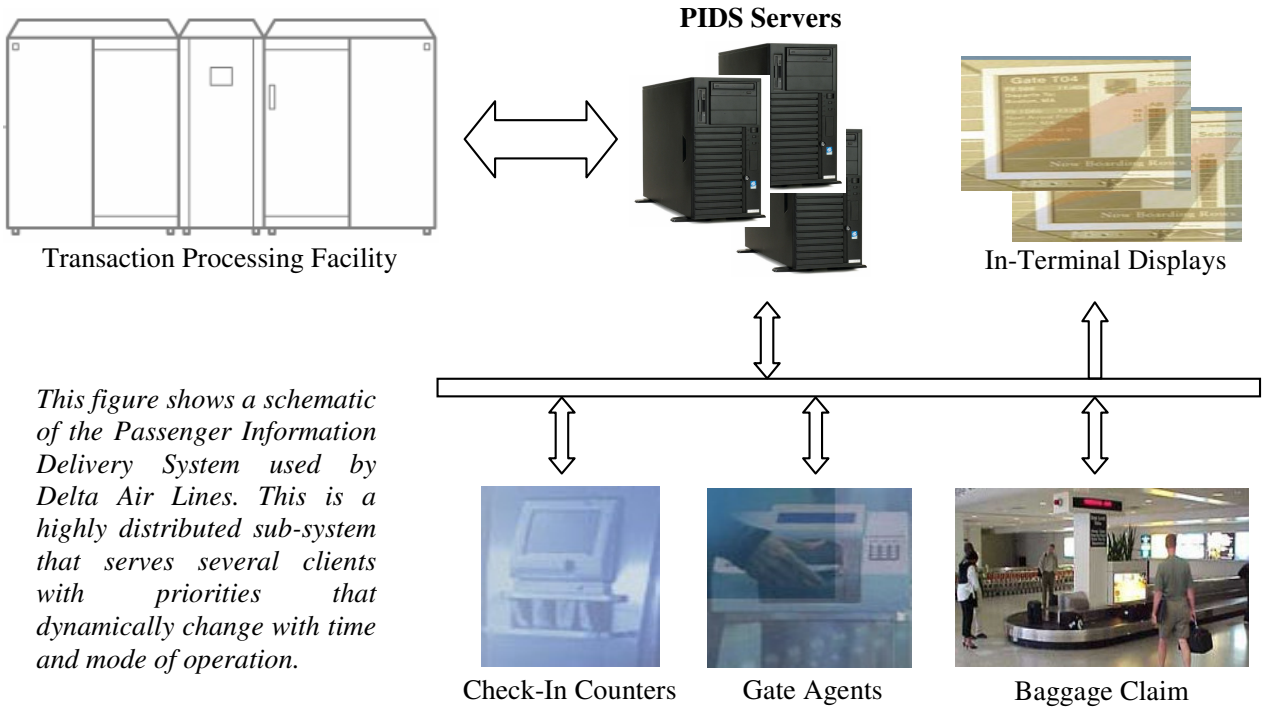
- *Trust* - An important criterion for making an acceptable self-management solution is to expose the reasoning behind the adaptation actions, and to provide an interface using which the degree of self-management can be fine-tuned. To address this issue, we build system models which are human understandable and we associate each policy with a confidence attribute that acts as the thresholding knob for controlling the degree of self-management.

In the following section, we describe the information we gathered about an operational information system deployed by Delta Air Lines, one of our industry partners. Our interaction with the administrators and developers exposed us to the complex, globally distributed information system and motivated us to examine the reasons for the lack of self-managing capabilities and to think of solutions for such systems.

1.1 Motivating Example

The Passenger Information Delivery System – PIDS (shown in Figure 1) – is a middleware developed at Delta Technology, Inc. to serve two important needs of the airline. First, it is responsible for managing the passenger data (sourcing the data from TPF). Second, it provides access to passenger information via events and service interfaces. The PIDS middleware, which according to estimates by Delta Technology processes around 9.5 billion events annually, ensures near real-time delivery of processed events to 'consumers' – programs that need to receive the events – and to a database of current booking and flight information used in activities like those in support of Delta's web site. PIDS collects data from all over the airline. While much of its information comes out of the airline's TPF-based Deltamatic Reservation and Operational Support System (OSS), additional inputs like gate information, information about weather, etc. arrive from airports throughout Delta's worldwide system. Further, passenger information is provided by the reservation system. Finally, most planes generate and transmit their own landing time, which is provided to PIDS via FPES (the flight progress event system).

There are hundreds of variables associated with the PIDS system that represent its operational state. Some of these variables are enumerated in Table 1. A system administrator manages the system by virtue of having the ability to modify some of these variables, examples including the number of client service threads or the number of workflow service threads. More specifically, such modifications of state variables constitute the set of actions allowed for managing the system. The actions of a system administrator to respond to an event (like increased workflow processing delay) are based on his wisdom (mental model of the system behavior) and the prevailing conditions (value of different variables representing the current state). In the following sections we will try to formalize these observations. There are several goals associated with the PIDS system and some of these may actually conflict with each other - like '*the system should not introduce more than 1 second of latency in its processing workflow*' and '*the system should be able to support 3000 concurrent cache access requests from the*



This figure shows a schematic of the Passenger Information Delivery System used by Delta Air Lines. This is a highly distributed sub-system that serves several clients with priorities that dynamically change with time and mode of operation.

Figure 1. Some Interactions in the PIDS Middleware System

clients'. Note that it is possible that supporting more clients might lead to violation of the latency requirements.

1.2 Related Work

Policy Research. There has been a lot work in the domain of using policies for simplifying the management tasks associated with system administration. Over the last decade, researchers, both in academia and industry, have focused on issues like policy specification languages [8], frameworks [4, 19] and toolkits [17]. The research presented in this paper builds on the work done in the above mentioned areas and is a logical next-step, as the focus is on applying the policy-research to the management intensive domain of enterprise-scale systems. The policy research in the domain of automated network management that deals with issues like security, access control and other associated management tasks [20, 18] justifies our stand on studying the impact and application of policy research to another rich domain. More recently, researchers have started evaluating the pros and cons of applying policy research for managing IT systems at large business enterprises [6]. This research, which is in its nascent stages, promises to provide systems that will manage themselves in accordance to high-level business goals [1]. The issues concerning human expertise and policy representation have also been explored in a recent paper [13].

Autonomic Computing & Self-Managing Systems. The task of implementing self-managing systems is a multi-step process in which policies can play an important role. Policies are a way to dictate the behavior of a self-managing system. This is in line with the vision of autonomic computing - 'to design computing system that can manage themselves given high-level objectives from administrators' - as described in [14]. There has been a lot of work in the domain of enabling self-management for a wide variety of systems. The SLA-based approach to manage systems has been explored by number of researchers [21]. In our prior work, we had focused on enabling self-management capabilities for distributed data stream systems [16, 15]. Some researchers have also explored the use of rule-based self-management approach for managing applications [3]. The use of utility-functions for self-management has also been explored in specific reference to event-based systems [5] and an interesting take on aggregate utility-functions is presented in [2]. It turns out that defining utility-functions for enterprise-scale applications is a tough task because it may not be possible to mathematically model all the factors that can potentially effect the state of the enterprise system.

Bayesian Networks. Bayesian networks or the Belief networks have found applicability in a number of AI domains and they represent one of the best classification tools available to researchers. A tutorial on Bayesian networks is presented in [11]. Several specializations of the Bayesian net-

Table 1. Some variables associated with the PIDS middleware

Variable	Description
E2EL	The end-to-end latency introduced by the processing workflow.
ELPP	The average queuing delay at individual PIDS processing nodes.
CLIE	Number of cache access clients being served at any time.
ETTR	Expected time to recover from a failure.
EDRR	Events dropped in last 100,00,000 events.
CSTH	Client service threads at individual PIDS processing nodes.
WSTH	Workflow service threads at individual PIDS processing nodes.

works have been proposed in literature the most important ones being the Naive Bayes [9] and the Tree Augmented Naive Bayes or the TANs [10]. In reference to applying Bayesian networks for modeling computer systems, a very innovative approach for correlating instrumentation data to system states is presented in [7].

1.3 Road Map

The rest of this paper is organized as follows. In Section 2 we briefly describe the system state-space model followed by an overview of the requirement for enabling policies in a computer system. Section 3 describes our approach in detail and focuses on the three main aspects of our approach - the partitioning algorithm, the model building techniques and the confidence attribute. Implementation details are presented in Section 4. In Section 5 we present the evaluation of our techniques using the implementation of our partitioning algorithm and the simulator. We finally conclude in Section 6 with some open problems for further research in this area.

2 System Overview

In this section we present a formal model for describing the system state-space. We identify the kind of variables that can possibly exist in a system state-space. The formal model is used in the following sub-section which deals with policy enablement.

2.1 System State-Space Model

We consider a system whose state can be represented by a set $\mathcal{V}$ of n variables $\{v_1, ..., v_n\}$, which are not necessarily independent. Out of these n variables the systems operational status (like failed, stable, unstable, etc.) can be determined by using only a subset $\mathcal{V}_\phi$ (an example of such variable would be the delay experienced by the users of an enterprise's website) of the state variables $\mathcal{V}$. Therefore, $\mathcal{V}_\phi$ is the set of variables of interest as far as the system's operational status is concerned. We also note that some variables (e.g. the length of input buffer at server nodes) have actions associated with them, using which the variable can be deterministically modified. We denote the set of such variables using $\mathcal{V}_\alpha$ which is a subset of $\mathcal{V}$, and $\mathcal{A} = \{(a_i, v_i^\alpha) | v_i^\alpha \in \mathcal{V}_\alpha\}$ represents the action and variable associations. The variables in $\mathcal{V}_\alpha$ are said to be deterministically modifiable because it is assumed that we know the effect of applying a_i on the corresponding variable v_i. More formally, if Ω represents the application of an action (or an ordered list of actions) on a variable (or a set of variables), then $\Omega(a_i, v_i)$ is supposed to be known for a given system state. Now, if $\mathcal{L}$ represents an ordered list of actions (possibly repeated) from the set A. Then $\mathcal{V}$ is closed under the operation Ω for any possible member of $\mathcal{L}$. Mathematically, this implies the following formulation

$$\Omega(L_1, V_1) = V_2 \tag{1}$$

where L_1 is an instance of $\mathcal{L}$ and V_1, V_2 are instances of $\mathcal{V}$.

The set of variables $\mathcal{V} - (\mathcal{V}_\phi \cup \mathcal{V}_\alpha)$ play an important role in determining the effect of any action and are not redundant. An example of such variable would be a measurement of number of disk-operations - such a variable is usually not a member of $\mathcal{V}_\phi$, which is used to determine acceptable system operational status; and this metric, in general, cannot be deterministically affected by allowed system actions (e.g. allocating another disk-array). However, such variables play an important role (as shown in Section 3.1) in determining the action to be taken in response to system events. The above discussion leads us to a formal definition of a manageable system.

Definition - *A system is said to be manageable if for all possible and valid instances V_1, V_2 of $\mathcal{V}$ there exists an instance L_1 of $\mathcal{L}$ such that $\Omega(L_1, V_1) = V_2$.*

The variables in V_α are related in some way to the variables in V_ϕ. Our aim is to discover a function χ that maps the space of variables of interest (V_ϕ) to the space of actionable variables (V_α). Note that the function χ may depend on the variables in $V - (V_\phi \cup V_\alpha)$.

To put the above discussion in perspective, such a system model can be readily applied to the example discussed

in Section 1.1. For example, the list of variables, enumerated in Table 1, constitute the set $\mathcal{V}$ of state variables for the PIDS system. The set of variables $\{E2EL, CLIE\}$ are the variables of interest for determining the acceptable or unacceptable system state and therefore constitute the set $\mathcal{V}_\phi$. The set $\mathcal{V}_\alpha = \{CSTH, WSTH\}$ constitutes the set of variables that have action associations. Finally, based to our definition, the trivially represented PIDS system is manageable because there exists an ordered list of actions $L \in \mathcal{L}$ which can translate the PIDS system from one operational state, V_1 (possibly unacceptable) to another operational state, V_2 (possibly acceptable).

2.2 Enabling Policies

There is a substantial amount of effort involved in making a system ready for policy-based management. Firstly, the system should be able to measure and export the current value of variables that constitute the state-space for the system. One can think of this as 'dials' on a control dashboard used for managing a very large system. Secondly, the ability to modify some of the variables is also central to the idea of policy enablement. One can similarly think of this capability as the 'knobs', which can deterministically change the value displayed on some 'dials'. In terms of our system state-space model, the variables represented by 'dials' are the variables in the set $\mathcal{V}$. The variables which have an associated 'knob' constitute the set $\mathcal{V}_\alpha$. The 'knobs' can in turn be used to take actions specified using $\Omega(a_i, v_i^\alpha)$.

In order the enable policies in a policy-ready system we need to have a way for representing the policies, mechanisms that discover and learn policies at runtime (note that this will require us to determine the function χ that essentially encapsulates the system model), ways to enforce policies and techniques for keeping the policies updated for the current system environment. The following subsections briefly describe our approach to handling these issues. Some of these issues will later be dealt in detail in Section 3

Policy Specification - We use a modified form of the well accepted event-condition-action (ECA) format for specifying the policies. The ECA specification is very useful when it comes to enforcing policies for any system. We however, extend the specification to include a confidence-attribute that is based on the amount of evidence on which the policy is based. The *event* in our policy description is a change in the value of some variable(s) in $\mathcal{V}_\phi$. The *condition* that triggers the action associated with the policy is specified over the set of variables in $\mathcal{V}$. The *action* is similarly specified as the modification in the value of some variables contained in V_α as determined using the function χ. The confidence-attribute is matched against a system-wide threshold set by the system administrator to check if a particular policy is enforceable or not, thereby giving the administrator a control over the degree of self-management.

Policy Discovery - We believe that all policies cannot be specified and that the system may need to discover some policies on the fly. We use a novel state-space partitioning scheme, described in Section 3.1 to first reduce the system state-space under consideration at any instant. Then for each partition we make use of greedy algorithm to discover the most important variables from the set V_α (i.e. the right knobs). We finally make use of bayesian networks to model the state-space contained in each partition, enabling us to find the values to which the 'knobs' should be adjusted to. We elaborate on these techniques in the following sections.

Policy Enforcement - The interfaces that export the current value of system variables are continuously monitored for any changes. These changes in the value of some variables may cause some policy to evaluate its condition and if the condition evaluates to true the action specified as part of the policy is taken. In simple words, when a problem occurs (i.e. the value on some dials signals something bad) the self-management subsystem tries to (1) find the 'right-knobs' and then (2) adjusts them to some appropriate new values.

Policy Refinement - Policies that are either specified or are learnt by the system may need to be changed because the conditions under which such policies are valid may change with time. An instance of this would be the addition of more nodes to the network underlying the operational information system. Such instances may lead to changes in threshold values that trigger an action specified as part of the policy. Our techniques are able to keep track of such changes in the environment and in response, they suitably modify the policies.

3 Solution Approach

The system state-space model proposed in Section 2.1 defines a manageable system and once $\mathcal{V}$, $\mathcal{V}_\phi$, $\mathcal{V}_\alpha$, $\mathcal{A}$ and χ are known for a system, an adminitrator or the self-management framework can easily navigate the system from one state to another. We assume that the sets $\mathcal{V}$, $\mathcal{V}_\phi$, $\mathcal{V}_\alpha$, $\mathcal{A}$ are known apriori. This implies that the system variables, the variables of interest and the deterministically modifiable variables along with the ways to modify them are known. This is true for distributed systems where the system variables like number of network nodes, link capacities, etc. are known, similarly the variables of interest like end-to-end delay are also a known value and lastly one knows of variables like allocated buffer-length at network nodes which can be deterministically modified by changing some system parameters. The problem is to find the function χ, and this means that we need to find a way to model the system. Remember that the function χ relates the vari-

ables in $\mathcal{V}_\phi$ to the variables in $\mathcal{V}_\alpha$ and the function χ can change for different values of variables in $\mathcal{V} - (\mathcal{V}_\phi \cup \mathcal{V}_\alpha)$. Once the function χ has been determined for the system state-space, one can easily find and/or adapt the actions that form part of the policy specification.

However, building a model (i.e. determining χ) for understanding the behavior of an enterprise-scale system is a tough task. This can be attributed to the fact that in such systems there are a large number of variables (e.g., bandwidth, workload, queue length at servers, etc.), each one of which can potentially affect the state of the system and more often than not these variables also interact amongst themselves. For example, in a certain sub-space of the system's state-space the bandwidth between participating nodes may be the bottleneck and any modification to the priority of processes may have little or no effect to the observed performance. The situation may similarly be reverse for some other system state sub-space where server capacity may be the bottleneck and any modification to the inter-node bandwidths may have no effect on the observed performance. The two insights that follow from the above example are that -

1. Finding a single function to model the entire state-space of an enterprise-scale system might lead to very crude and incorrect system models.

2. There exists system state sub-spaces where the effect of certain variables can essentially be ignored from the system model.

The above discussion motivates the need to partition the system state-space. The following sub-section elaborates on the specific requirements for the partitioning scheme.

3.1 System State-Space Partitioning

The aim of our partitioning scheme is to create system state-space partitions such that -

- the involved system variables exhibit some homogeneity in their behavior inside the partition, which is beneficial for building the system model.

- the number of 'knobs' required to manage the system within the partition is minimized, which is beneficial for the purpose of learning and adapting the actions specified as part of policy.

To incorporate the concept of partition homogeneity we create partitions such that operational states contained in the partition are close to each other. Note that partition homogeneity corresponds to macro-level states of the system, for instance in one partition the underlying network may be the bottleneck (making server capacity redundant) while in some other partition the server capacity may be the bottleneck (similarly making the network capacity redundant). In order to minimize the 'knobs' we want to ensure the partitions are created such that the 'knobs' needed in one partition are possibly not needed in the other. This corresponds to making partitions which are orthogonal to each other. The partitioning algorithm employed by our framework is described next.

3.1.1 The Partitioning Algorithm

Let, S be the operational states contained in the initial system state space for which $\mathcal{A}$ defines the association of actions with the variables in $\mathcal{V}_\alpha$. For simplicity the discussion here assumes that it is possible to define a measure of normalized distance between any two operational states. Techniques for doing such operations exist and interested readers may refer to well-known techniques like Mahalanobis distance. We define an operator $\delta_\mathcal{V}$ over a pair of operational states from a partition, which finds the normalized distance between the two operational states considering only the dimensions contained in the set $\mathcal{V}$. We also define the operation θ over a pair of operational states from a partition. The operation θ finds the number of places in which the two states differ, considering only the dimensions corresponding to the set $\mathcal{V}_\alpha$ for the partition under investigation. Finally, we define

$$\upsilon(s_1, s_2) = \eta \times \delta_\mathcal{V}(s_1, s_2) + \mu \times \theta(s_1, s_2) \qquad (2)$$

where, η and μ can take values from the range [0,1] and these are used to configure υ for weighted distance and orthogonality.

To evaluate if we need to partition a given system state-space P, we try find a subset $\mathcal{V}'_\alpha$ of $\mathcal{V}_\alpha$ such that

$$\sum_{\forall s_i, s_j \in S} \delta_{\mathcal{V}_\alpha - \mathcal{V}'_\alpha}(s_i, s_j) \leq \Delta_{max} \qquad (3)$$

$$|\mathcal{V}'_\alpha| \leq f \qquad (4)$$

where Δ_{max} is a user defined parameter that represents the maximum allowed representation error for the actionable variables and f represents the maximum number of actions that can be used to manage the system in the given partition. We employ a greedy approach for finding $\mathcal{V}'_\alpha$, i.e. we add the member of $\mathcal{V}_\alpha$ to $\mathcal{V}'_\alpha$ which causes the greatest reduction in the L.H.S. of the equation 3. We repeat the above process until the L.H.S. becomes lesser than Δ_{max}, at this point we look at the cardinality of the set $\mathcal{V}'_\alpha$ - if the cardinality is less than f we do not partition the system state-space, otherwise we proceed to partition the system state-space. The $\mathcal{V}'_\alpha$ so determined becomes the $\mathcal{V}_\alpha$ for the partition.

We start by finding a pair of states s_1 and s_2 from the set of all such pairs contained in the set S such that $\upsilon(s_1, s_2)$

is maximized. The pair s_1 and s_2 acts as the seed for the two new system state sub-spaces S_1 and S_2 that will be created. We then iterate through the remaining operational states in the set S, adding the operational state s_i to S_1 if $\delta_{\mathcal{V}}(s_i, s_1) \leq \delta_{\mathcal{V}}(s_i, s_2)$. One can alternatively use the centroid of existing operational states in the evolving partitions to determine the membership. Once the two new partitions S_1 and S_2 have been created, we find the set $\mathcal{V}_\alpha$ for them using the greedy approach described above. If the criteria defined by Δ_{max} and f is not met by any partition then we repeat the above scheme for that partition. We now enumerate the advantages of the partitioning scheme for the purpose of enabling policy-driven self-management.

- *Simplifies Policy Learning*. Our approach intelligently reduces the space of possible actions that could be taken in response to an event. This greatly simplifies the process of correlating the events to actions for the purpose of determining ECA policies.
- *Assists in Problem Diagnosis*. The system might migrate through a series of system state 'partitions' before ending in an unacceptable state (e.g. SLA violation). The path followed by a system before a failure may contain information about the events that may have led to a failure, and can therefore assist in problem diagnosis and constructing complex policies.
- *Simplifies Problem Resolution*. If a system enters an unacceptable state during its operation. The model corresponding to the partition to which this unacceptable state belongs can possibly be used to arrive at a resolution to the problem.
- *Reducing Monitoring Overhead*. The partitions that are created by our algorithm allow us to ignore a subset of variables when the system is operating in that partition. This can potentially allow us to monitor such variables at reduced frequencies. However, we have not fully explored this possibility.

Once the system state-space has been partitioned we build a system *micro-model* corresponding to each partitioned sub-space. A system model in our framework consists of several micro-models each one of which models a sub-space of possible system states. The micro-model to be applied is determined based on the current state of the system. Since, we attempt to model only a small partition of the entire system state-space at a time we are able to build models even for systems with a very high number of variables. This makes our approach highly scalable.

3.2 Building System Micro-Models

To build system *micro-models* we make use of a special class of Bayesian Networks called the Tree Augmented Naive Bayes or TANs. TANs have earlier been used to model system behavior for automated problem diagnosis and localization [7]. The main advantage of using bayesian networks (or its variants) is that their representation provides and easy way to inspect the relationships between the involved variables. This allows an expert to embed his *wisdom* into the self-management framework by proposing initial model, which can be further *refined* using learning techniques. Furthermore, by simple inspection an expert can single out any faults in the learnt system model.

A Bayesian network is represented as an acyclic graph whose vertices encode random variables and the edges represent statistical dependence relations among the variables and local probability distributions for each variable given values of its parents. In general, a Bayesian network is computationally very costly to build (i.e. if all dependencies are evaluated). TANs offer a model that is computationally cheap to build but at the same time it is sufficiently rich to encode various dependencies amongst the system variables. TANs are an improvement over the naive Bayesian networks which assume that all variables are fully independent for the purpose of building models. To put this in context, a naive Bayesian network would assume that all system variables (like end-to-end delay, priority, queue-length, etc.) are independent of each other. Thereafter, this assumption is used to classify the state of the system (like stable, unstable, optimal, etc.). However, naive Bayesian networks fail to capture dependencies amongst the variables which can be captured (to some extent) by making use of TANs. TANs impose a tree structured dependence graph between the variables such that each variable can have at most one parent other than the final variable of interest which determines the systems operational status and is determined by making use of the variables in V_ϕ.

To induce a model for a system using TAN one needs a set of observations that record system variables and the corresponding system state from an operational system. These observations are divided into the training set and the testing set. Given a set of variables there can be multiple TANs that can be formed and the problem is to find an optimal TAN given the set of observations. This is where wisdom can play an important role - a system administrator can prespecify some dependencies (like queue length is definitely dependent on event rate) and can similarly rule out certain dependencies. This greatly reduces the cost of inducing TAN model for the system.

3.3 Policy Confidence Attribute

The confidence-attribute associated with each policy helps us to deal with the issue of administrators *trust* in our self-management framework. The confidence attribute that is associated with each learned policy is a result of the way

Bayesian networks operate and the way the ECA policies are specified. The 'event' in a policy is responsible for invoking the policy, the 'conditions' then dictate the kind of 'action' and if the 'action' should be taken in response to the event. Applying the above to our state-space framework, the conditions are defined as some boolean operation represented as $C(v_1, ..., v_p)$, over the set of m variables that are contained in $\mathcal{V}$. The 'action' is represented as assignment of some new values to certain system variables that belong to $\mathcal{V}_\alpha$. Let the 'action' be represented as the assignment of new values to p variables $(v_\alpha^{i_1} = c_1, ..., v_\alpha^{i_m} = c_m)$. The partition for which the condition C evaluates to true is the model that is chosen. The model is then used to answer the question - *Given the system is in the goal state V_ϕ^{goal}. What is the probability that the set $\mathcal{V}_\alpha$ takes the values specified as part of the 'action'? The variables whose values are not specified as part of the 'action' are assumed to maintain* status quo. Therefore, if V_α^{now} represents the current operational values corresponding to the set $\mathcal{V}_\alpha$, the new operational values V_α^{policy} are obtained by assigning the values from the action specification to the corresponding variables. Given the chosen system model, V_ϕ^{goal} and V_α^{policy}, the confidence-attribute κ is defined as -

$$\begin{aligned} \kappa &= \Pr(V_\alpha^{policy} | V_\phi^{goal}) \\ &= \frac{\Pr(V_\alpha^{policy}, V_\phi^{goal})}{\Pr(V_\phi^{goal})} \end{aligned}$$

The system administrator can declare a threshold value $\kappa_{threshold}$ to have control over the policies that will be enforced. Only the policies for which $\kappa \geq \kappa_{threshold}$ will be autonomically enforced by the self-management framework.

4 Implementation

We have implemented the system state-space partitioning algorithm in C++, which takes as input a set of data which contains actual observed values from a running (or simulated) system and the index of variables which can be modified deterministically. The algorithm is currently implemented to accept only numeric values for the observed data. The user also needs to provide values for the two parameters η and μ as defined in Section 3.1.1. The output from the partitioning algorithm is the set of partitions and the corresponding index of variables which can be modified deterministically. The output is generated in the well-known C4.5 format to facilitate further processing. We then make use of jBNC [12], a java based implementation for the Bayesian networks to find a TAN corresponding to each partition. The TAN is then used for finding corrective actions, policy refinement, etc.

In order to evaluate our policy-driven self-management implementation we needed a large system with a huge number of variables that could be observed (sensors) and a set of variables that could be deterministically modified (actuators) to manage the system. Many of the systems available to us in our lab environment (like RuBiS and an implementation of industrial middleware from Delta Technologies) were not instrumented well enough to sense and actuate a sufficiently large set of variables, and often changing any environment parameter required restarting the application for the changes to take effect.

In order to overcome the problems mentioned above we decided to design a well instrumented simulator for simulating a large system implementing service oriented architecture. An implementation of service oriented architecture contains a set of services running on a distributed network of nodes that can be invoked by sending a message to the service, messages may or may not be generated as a result and if the messages are generated they may be forwarded to the source, to a sink or to some other service(s). Our implementation consists of four main components

- Server - A server represents a processing facility with a limited number of cycles per second, a limited memory, connections to other servers, ability to throttle server frequency at the expense of more power, etc.

- Service - A service represents the software which accepts certain types of messages, possibly generates some messages in response, determines the server cycles that will be used to process a certain message type given the available memory. There may be more than one service running on a server and they may have different priorities.

- Network-Link - A network link contains the bandwidth, delay, and cost per unit of data transmitted as attributes. A server may have multiple routes to the same destination.

- Client - A client represents a source or a sink for the messages. An event source'can have a rate of generating events that can vary with time. A sink measures the incoming event-rate, the average delay for update propagation and the current delay measured over a recent window.

A simulation can be started by adding a certain number of servers, some inter-server links, some services for processing messages and some clients. The simulation is run for a pre-specified amount of time and it dumps the state at configurable regular intervals. We make use of these state dumps to evaluate our algorithms.

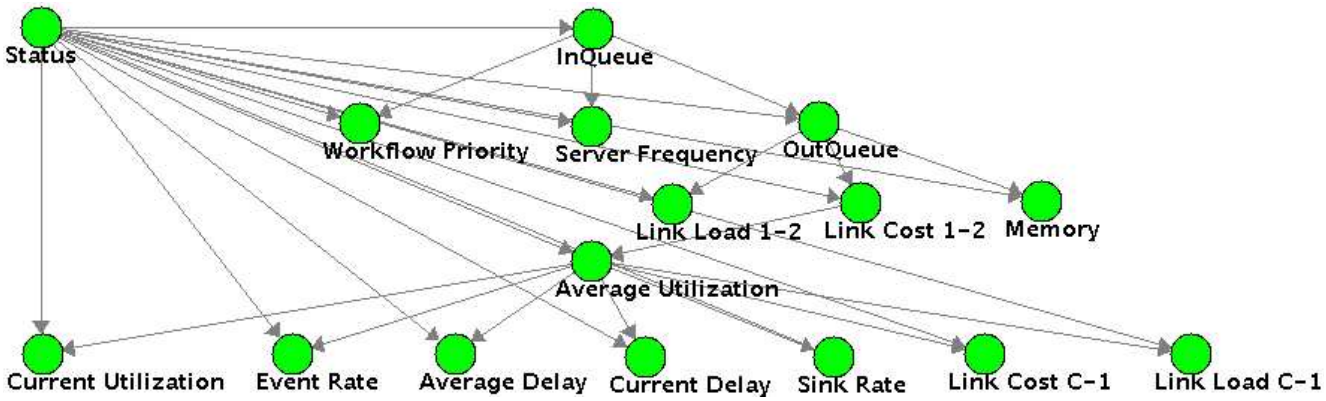


Figure 2. TAN model resulting from a simple simulation

5 Experiments

In this section, we report some preliminary evaluation that we conducted for our self-management framework. We first ran a simple simulation that consisted of only one server, one source client and one sink client connected to the server. Thereafter we setup a simulation with 8 servers, 11 source clients, 10 sink clients and 11 services. The simulation output consisted of 147 variable dump at each monitoring checkpoint. We evaluated the effect of our partitioning scheme on the inferencing capabilities of our TAN models.

5.1 Simple Simulation

We wanted to get the feel of the TAN models and if they were intuitive. This is important because we wanted a system administrator to be able to look at these models and point out anything that is not in line with the 'common wisdom'. We had to resort to a simple simulation that dumped 16 variables at each time-step to validate the model, which might have not been possible if we had built a model with hundreds of variables. The resulting TAN model is shown in Figure 2, which is based on a half-an hour run of the simulator dumping data every 30 seconds. There are very intuitive patterns in the figure, like the dependency between the input queue size and the output queue size. Similarly observe the dependency between the server frequency and the input queue size.

5.2 Complex Simulation

We ran the 8 server simulation for 1 hour. The simulation dumped the 147 state variables every 30 seconds. During the course of simulation we kept modifying the system conditions like the event rates from the sources, modifying the server frequencies, using alternate high or low-cost links to the destination and changing the priorities for various services running at a server. We also ran the simulation for another 10 minutes, dumping data at every 30 seconds to test the generated models against. We constructed the TAN model using the original data and ran against our test data to find the accuracy of the generated model. The partitioning algorithm divided the set of observations into 3 sets and a micro-model was generated for each of them. In order to test against our micro-models we directed the test data to the micro-model for which the V_α from test data state was closest to the cluster seed on which the micro-model was based. The results reported in Table 2 show that specialized micro-models work better than a single model when the whole data-set is considered. However, we notice that for one of the partition the resulting micro-model performs worse than the single model. This can be attributed to the type of data that was collected for this experiment, which probably did not contain enough evidence for the micro-model of partition-1. In general the micro-models performed better with the test data as the partitioning was aimed to find sub-spaces in the system state-space where the behavior of the system is almost homogeneous.

Table 2. Comparison between the accuracies (in %age) of single and micro-models

Data Type	Single Model	Micro-Model
Whole Data	92.3	95.4
Partition-1	94.5	93.9
Partition-2	91.2	96.3
Partition-3	91.9	94.8

6 Conclusions & Future Work

In this paper we have presented an overview of the approach that can be used for enabling policy-driven self-management in enterprise-scale systems. There are several issues that need to be addressed before such a framework

can become useful for managing real-world applications. These issues include the monitoring overhead that may be imposed by the mechanism used to enable policies. Moreover, it might not be possible to monitor the complete set of variables in several systems. We are still working to refine our system and this work contains only the preliminary results that we have obtained using our prototype implementations.

References

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

[2] A. AuYoung, L. Grit, J. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *International Symposium on High Performance Distributed Computing, HPDC*, 2006.

[3] V. Bhat, M. Parashar, H.Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *International Conference on Autonomic Computing, ICAC*, 2006.

[4] M. Bhide, A. Gupta, M. Joshi, M. Mohania, and S. Raman. Policy framework for autonomic data management. *IEEE International Conference on Autonomic Computing, ICAC*, 2004.

[5] S. Bhola, M. Astley, R. Saccone, and M. Ward. Utility-aware resource allocation in an event processing system. *IEEE International Conference on Autonomic Computing, ICAC*, 2006.

[6] Z. Cai, V. Kumar, B. F. Cooper, G. Eisenhauer, K. Schwan, and R. E. Strom. Utility-driven management of availability in enterprise-scale information flows. *ACM/IFIP/USENIX 7th International Middleware Conference*, 2006.

[7] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *International Symposium on Operating Systems Design and Implementation, OSDI*, 2004.

[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995, 2001.

[9] P. Domingos and M. J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[10] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[11] D. Heckerman. A tutorial on learning with bayesian networks, 1995.

[12] jBNC: Bayesian network classifier toolbox. `http://jbnc.sourceforge.net/`, as viewed on 03/18/2006.

[13] E. Kandogan, C. Campbell, P. Khooshabeh, J. Bailey, and P. Maglio. Policy-based management of an e-commerce business simulation: An experimental study. In *International Conference on Autonomic Computing, ICAC*, 2006.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[15] V. Kumar, Z. Cai, B. F. Cooper, G. Eisenhauer, K. Schwan, M. Mansour, B. Seshasayee, and P. Widener. Implementing diverse messaging models with self-managing properties using iflow. *3rd IEEE International Conference on Autonomic Computing, ICAC*, 2006.

[16] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. *2nd IEEE International Conference on Autonomic Computing, ICAC*, 2005.

[17] L. Lymberopoulos, E. C. Lupu, and M. S. Sloman. Ponder policy implementation and validation in a cim and differentiated services framework. *Network Operation and Management Symposium, NOMS*, 2004.

[18] N. Minsky. A scalable mechanism for communal access control. *Conference on New Challenges for Access Control, NCAC*, 2005.

[19] Web services policy framework. `http://www-128.ibm.com/developerworks/library/specification/ws-polfram/%`, as viewed on 03/18/2006.

[20] M. J. Wright. Using policies for effective network management. *International Journal of Network Management*, 9(2):118–125, 1999.

[21] L. Zhang and D. Ardagna. Sla based profit optimization in autonomic computing systems. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 173–182, New York, NY, USA, 2004. ACM Press.