Modelling Reinforcement Learning in Policy-driven Autonomic Management

Raphael M. Bahati and Michael A. Bauer
Department of Computer Science
The University of Western Ontario, London, ON N6A 5B7, CANADA
Email: {rbahati;bauer}@csd.uwo.ca

Abstract

Management of today's systems is becoming increasingly complex due to the heterogeneous nature of the infrastructure under which they operate and what the users of these systems expect. Our interest is in the development of mechanisms for automating the management of such systems to enable efficient operation of systems and the utilization of services. Central to autonomic management is the need for systems to monitor, evaluate, and adapt their own behavior to meet the different, and at times seemingly competing, objectives. Policy-driven management offers significant benefit to this effect since the use of policies can make it more straightforward to define and modify systems behavior at run-time, through policy manipulation, rather than through re-engineering. This work examines the effectiveness of Reinforcement Learning methodologies in determining how to best use a set of active (enabled) policies to meet different performance objectives. We believe that Reinforcement Learning offers significant potential benefits, particularly in the ability to modify existing policies, learn new policies, or even ignore some policies when past experience shows it is prudent to do so. Our work is presented in the context of an adaptive policy-driven autonomic management system. The learning approach is based on the analysis of past experience of the system in the use of policies to dynamically adapt the choice of policy actions for adjusting applications and system tuning parameters in response to policy violations. We illustrate the impact of the adaptation strategies on the behavior of a multi-tiered Web server consisting of Linux, Apache, PHP, and MySQL.

Index Terms—Autonomic Management, Reinforcement Learning, Policy-driven Management, QoS Provisioning.

1 Introduction

Today's Information Technology (IT) infrastructure is becoming heterogeneous and complex to the point that it is extremely difficult, if not impossible, for human operators to effectively manage. Increasingly, the combination of applications integrated within a single or multi-computer environment has become a key component in the way many organizations deliver their services and provide support. Ensuring that such systems meet the expected performance and behavioral needs is among the key challenges facing today's IT community. To this end, there has been a lot of interest in the use of explicit system performance models to capture systems behavior as well as provide guidance in managing applications and systems. While these approaches have achieved some success in specific areas, we note that developing models that accurately capture systems dynamics, particularly for the state of the enterprise systems, is highly nontrivial.

Our interest is in the development of policy-driven autonomic techniques for managing these types of systems. Required or desired behavior of systems and applications can be expressed in terms of policies. Policies can also be used to express possible management actions. As such, policies can be input to or embedded within the autonomic management elements of the system to provide the kinds of directives which an autonomic manager could make use of in order to meet operational requirements. The effective use of policies in autonomic management requires that the policies be captured and translated into actions within the autonomic system. As such, policies can provide the kinds of directives best suited for flexible, adaptive, and portable autonomic management solutions.

Previous work on the use of policies has mainly focused on the specification and use "as is" within systems and where changes to policies are only possible through manual intervention. In an environment where multiple sets of policies may exist, and where at run-time multiple policies may be violated, policy selection is often based on statically configured policy priorities which an administrative user may have to explicitly specify. As systems become more complex, however, relying on humans to encode rational behavior onto policies is definitely not the best way forward. It is imperative, therefore, that autonomic systems have mechanisms for adapting the use of policies in order to deal with not only the inherent human error, but also the changes in the configuration of the managed environment and the complexities due to unpredictability in workload characteristics.

Self-optimization describes the ability of autonomic systems to evaluate their own behavior and adapt it accordingly to improve performance [1]. In the context where policies are used to drive autonomic management, this may often require having a system monitor its own use of policies to learn which policy actions are most effective in encountered situations. The system might try to correlate management events, actions and outcomes based, for example, on the long-term experience with a set of active policies. This information could then be used to enable the system to learn from past experience, predict future actions and make appropriate trade-offs when selecting policy actions. The use of policies in this context offers significant benefits to autonomic systems in that it allows systems administrators to focus on the specification of the objectives, leaving it to systems to plan how to achieve them. This paper looks at how Reinforcement Learning methodologies could be used to guide this process. In particular, we demonstrate how a model derived from the enabled policies and the consequences of the actions taken by the autonomic system (which we first proposed in [2]) could be "learned" on-line and used to guide the choice of policy actions for adjusting system's tuning parameters in response to policy violations.

The rest of this paper is organized as follows. We begin with a background on Reinforcement Learning in Section 2. In Section 3, we describe the structure of the policies we assume in our work and provide examples illustrating how these policies are used to drive autonomic management. Section 4 presents an adaptive policy-driven autonomic management architecture illustrating key control feedback interactions involved in guiding the selection of policy actions for resolving Quality of Service (QoS) requirements violations. Section 5 and 6 describe how Reinforcement Learning methodologies could be used to model an autonomic computing problem involving QoS provisioning. Section 7 describes the prototype implementation of the learning mechanisms, illustrating the impact of the adaptation strategies on the behavior of a multi-tiered Web server. We review some related work in Section 8, and conclude with a discussion on key challenges and possible direction for future work in Section 9.

2 Reinforcement Learning Background

Reinforcement Learning describes a learning paradigm whereby, through trial-and-error interaction with its environment (see Figure 1), an agent learns how to best map situations to actions so as to maximize long-term benefit [3]. As such, Reinforcement Learning is often associated with training by reward and punishment whereby, for each action the agent chooses, a numeric reward is generated which indicates the desirability of the agent being in a particular state. A key distinction between Reinforcement Learning and other forms of learning is on what information is communicated to the learner after an action has been selected. In *supervised learning*, for example, the learner only has to visit a state once to know how to act optimally if it encounters the same state again. This is because, for each action taken, the learner is told what the correct action should have been. In Reinforcement Learning, on the other hand, the learner only receives a numeric reward which indicates how good the action was (as opposed to whether the action was the best in that situation). The only way for the learner to maximize this reward, therefore, is to discover which actions generate the most reward in a given state by trying them. Consequently, the learner is often faced with a dilemma: whether to use its current knowledge to select the best action to take, (*exploit*) or try actions it has not yet tried (*explore*) in order to improve its guesses in the future.

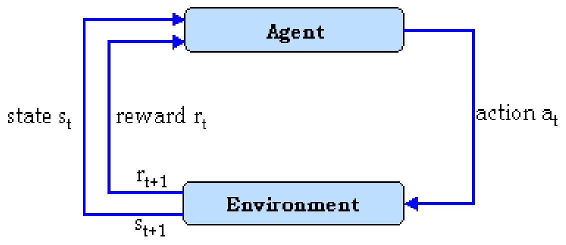


Figure 1. The agent-environment interaction in Reinforcement Learning [3].

As with many learning problems, it is often impractical to obtain an environment model that is both accurate and representative of all possible situations the learning agent may encounter while interacting with the environment [3]. While model-free Reinforcement Learning methods exist which are guaranteed to find optimal policies (i.e., choices of actions per situation), they make extremely inefficient use of data they gather [4]. One approach for overcoming this shortfall is for the agent to learn the model of the environment's dynamics, on-line, as it interacts with the environment. This has been demonstrated to significantly accelerate the learning process (see, for example, [5, 6, 7]). In this approach, a model is updated continually throughout the agent's lifetime: at each time step, the currently learned model is used for planning; i.e., using the learned model to improve the policy guiding the agent's interaction with the environment.

Several model-based learning algorithms exist in the literature and differ mainly on how the model *updates* are per-

Algorithm 1 Dyna-Q

```
Input: Initialize Model$(s,a)$ for all $s \in S$ and $a \in A(s)$
1: for $i = 1$ to $\infty$ do
2:     $s \leftarrow$ current (non terminal) state
3:     $a \leftarrow \epsilon$-greedy$(s, Q)$
4:     Execute $a$; observe resultant state, $s'$, and reward $r$
5:     $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
6:     Model$(s,a) \leftarrow s', r$
7:     for $j = 1$ to $k$ do
8:         $s \leftarrow$ random previously observed state
9:         $a \leftarrow$ random action previously taken in $s$
10:        $s', r \leftarrow$ Model$(s,a)$
11:        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
12:    end for
13: end for
```

formed. In this paper, we make use of an algorithm called Dyna-Q [8] (see Algorithm 1) which estimates action-values; i.e., a measure of how good it is for an agent to perform a particular action in a given situation. Briefly, the algorithm works as follows: Beginning with state s, the agent selects action $a \in A(s)$ and observes the resultant state s' and reward r. Using this information, the agent updates the action-value associated with action a (line 5) and adds this information to the current system model (line 6). It also performs k additional updates of the model by randomly selecting and updating the action-value estimates of k state-action pairs (lines 7 - 12). In the sections that follow, we describe how *model-learning* mechanisms could be applied to an autonomic computing problem involving QoS provisioning. But first, we begin with a look at how policies could be used to drive autonomic management.

3 Autonomic Management Policies

We have been exploring the use of policies as the basis for autonomic management, with a particular focus on e-commerce systems. We feel that policies can provide the kinds of directives which autonomic systems can and should rely on when making management decisions. As with much of the previous work on policy-driven management (see, for example, [9, 10]), our interest is on *action policies* (expressed as *obligation policies* in Ponder [9]) since they can be defined and modified on a per component basis and can provide useful information for autonomic managers. The use of action policies within autonomic computing is likely to continue partly due to their simplicity and, unlike *goal policies* [11, 12] and *utility policies* [11, 13, 14, 15], do not require a system model in order to be used [11]. In this work it is assumed that action policies are event-triggered, action-condition rules [9]. An event triggers the evaluation of a rule of the form "**if** [conditions] **then** [actions]". An event is generated as a result of some condition of the state of the system being true. This section looks at what these policies are and how they could be used within autonomic computing.

3.1 Policy Structure

We assume a policy to consist of several attributes including one or more conditions and an ordered list of actions that make adjustments to some tuning parameters:

3.1.1 Policy Rule

A policy rule basically consists of a *policy type* (discussed in Section 3.2), *policy name*, a *conditions set* which is dependent on one or more conditions, and an *actions set* (see, for example, Figure 3). Because a policy may apply to many different components, the assumption is that the policy would be instantiated at run-time, say when the management system starts its components or a particular application is started. For example, the policy *target* might be instantiated to a particular host within a network of hosts, that is, the same policy could apply to each of the hosts though each would be monitored separately. In the policy example of Figure 3, the policy target is the process corresponding to the Policy Enforcement Point (PEP). The policy *subject* is the management component that should receive the event when there is a violation. Hence, the subject would also be instantiated. In our prototype, the subject of an *expectation policy* (see Section 3.2.2) would likely be the process corresponding to the Policy Decision Point (PDP) as illustrated by the policy of Figure 3; in larger systems, there could be other management components to receive events or multiple PDPs. A policy has at least one other attribute which can change dynamically, which specifies whether a policy is enabled (set to true) or not.

3.1.2 Policy Condition

A policy condition captures the state of an application, a system, device, etc. It is assumed that events are generated from monitoring components and that the Event Handler (see Section 4.1) filters received events for those of "interest". An event specified by name only is essentially a Boolean value; i.e., the occurrence of the event itself is sufficient to take an action. An event with an attribute indicates that the value of the attribute is to be used in evaluating an expression, such as comparing the value to a threshold. It is also possible to have a policy which becomes violated only when multiple events or conditions occur. These are specified via the standard logical operators.

```
configuration  policy{InstallCPUMonitor(MonitorManager, localhost)}
 if(INSTALL:CPUMonitor = true)
 then{./CPUMonitor  test{IsConditionEnabled(CPU:utilization) = true}}
```

Figure 2. A configuration policy for installing a CPU Monitor.

3.1.3 Policy Action

A policy action defines what has to be executed should the condition(s) specified in the policy hold true. Each action is, essentially, the name of a function that should be executed. The function may have parameters that would be determined from the information associated with the policy, e.g., domain, events, event attributes, etc. One or more actions may also be specified. Each action may have an optional `test` associated with it, with or without parameters. The test can be used to determine if the component state or context invalidates the particular action. Such a test is a Boolean function or could return a value which is then compared to some threshold value. If the result of the test expression is "true" then that indicates that the action is enforceable; note that the negation of a test is permitted in which case the expression is "true" if the test evaluates to "false". As such, policy tests provide ways in which the degree of self-management could be controlled. Action sequences may be conjoined (i.e., "AND-ed" together) indicating that all the actions in the sequence should be executed. Alternative action sequences may also be specified in which case only one of the elements of the sequences would be selected.

In our current approach, we permit only a single action within a single expectation policy to be executed. This is done for two reasons. First, this is a strategy of "doing something simple" and seeing if there is a positive effect. If the change is not sufficient, then a violation is likely to occur again and a further action (which could be the same, e.g., increasing or decreasing the value of a parameter) can be taken. The management cycle in the implementation is short enough that this can happen quickly. Second, taking multiple actions makes it difficult to understand the impact of the actions; e.g., were they all necessary, were some more effective than others, etc. By having the autonomic manager take a single action and log that action and other information, an analysis component can examine that information and possibly determine which action(s), or the order thereof, is better, etc. We outline one such an approach in Section 6.

3.2 Policy Types

We are currently exploring the use of several types of policies for driving autonomic management.

3.2.1 Configuration Policies

Configuration policies describe those policies that are used to specify how to configure and install applications and services. This may include, for example, setting static configuration parameters based on the Service Level Agreement (SLA) requirements (e.g., performance, availability, quality of service), the given or expected environmental parameters (e.g., required services, number of active users), and the available resources (e.g., number of processors, processor speed, memory size, disk space).

A sample configuration policy for installing a CPU Monitor for the system of Figure 4 is shown in Figure 2. In this example, the `MonitorManager` (i.e., the poliy subject) is the component responsible for installing the CPU monitor on a `localhost` (i.e., the policy target). The policy test determines the conditions under which the CPU Monitor is to be installed. In this example, the monitor is installed only if the condition "`CPU:utilization`" is enabled - as determined by a set of enabled *expectation policies* (see Section 3.2.2). As such, changes to the policies driving autonomic management could also trigger dynamic reconfiguration of systems and applications. For example, by disabling the policy of Figure 9 (assuming, of course, that it is currently the only policy with a "`CPU:utilization`" condition), the CPU Monitor would be disabled as a result. A key advantage here is the reduction in the management overhead since events specific to CPU utilization would no longer be relevant when the policy is no longer active.

3.2.2 Expectation Policies

Expectation policies define information used to ensure that operational requirements are met and expected conditions not violated. We have also been using expectation policies to indicate how the system could optimize its use of resources. For example, a policy could indicate that, when the response time of requests to the server falls below a certain level, then Apache processes handling requests could be reduced. This would then free up system resources.

A sample expectation policy for resolving violations in Apache's response time is shown in Figure 3. It consists of two conjunctive conditions and three disjunctive actions. Note that the actions, which specify adjustments to the application's tuning parameters, are quite simple since each specifies a small - and in some cases the smallest pos-

```
expectation policy{RESPONSETIMEViolation(PDP, PEP)}
 if(APACHE:responseTime > 2000.0) & (APACHE:responseTimeTREND > 0.0)
 then{AdjustMaxClients(+25) test{newMaxClients < 151} |
      AdjustMaxKeepAliveRequests(−30) test{newMaxKeepAliveRequests > 1} |
      AdjustMaxBandwidth(−128) test{newMaxBandwidth > 255}}
```

Figure 3. A sample expectation policy for resolving Apache's response time violation.

sible - increment/decrement in the value of the parameter. For example, the Apache's `MaxClients` parameter could only be adjusted in increments/decrements of the number of threads per child process, as specified in the server's configuration. Thus, a general knowledge of how an increase/decrease in the value of a particular parameter impacts a system's performance metrics may be sufficient to define reasonable policies. For instance, a violation in Apache's response time could be due to the fact that there aren't enough server processes to handle clients requests, in which case increasing `MaxClients` could resolve the problem. If this is no longer possible (as determined by the action test), one might try to reduce the amount of time (i.e., `MaxKeepAliveRequests`) existing clients hold onto the server processes. And, if this is no longer possible, it could be that the server is overwhelmed by the number of requests in which throttling some may alleviate the problem. This is illustrated by the expectation policy in Figure 3. Since, only a single action could be executed, the order in which the actions are specified within the policy is also important. In this case, more drastic actions could be taken once it is no longer possible, for example, to meet the objectives through tuning application's parameters. This is precisely the purpose of the action "`AdjustMaxBandwidth(-128)`" which throttles requests to the server by reducing the rate at which the server processes clients requests based on a client's service class (see Section 7.3 for details).

3.2.3 Management Policies

Management policies deal with information and actions for managing the management system itself or for the overall administration of the system or applications. Such policies may include those for the prioritization of expectation policies, for diagnosis in determining an action, or involving some analysis (say of previous behavior) in determining an action. We are currently exploring the use of management policies to guide the on-line learning process. Our particular focus is on how the learning algorithms could be optimized to be less computational intensive in order to meet the resource constraints imposed by the environment. We comment further on this in Section 9.

4 System Architecture

A detailed view of the architecture for the adaptive policy-driven autonomic management system is depicted in Figure 4. Our approach to autonomic management involves providing quality of service support local to each host. Each local host, therefore, has a single Policy Decision Point (PDP) whose responsibility is to oversee the management of a single host according to the policies specified. In a multi-tiered Web-server environment, for example, several components (i.e., a Web server, an application server, and a database server) may cooperate to deliver a set of services. In the case that all these components are run on a single host, a local PDP will be responsible for ensuring that the managed application, as a whole, behaves as expected. Since each component would have it's own set of policies, more complex decisions regarding the choices of actions when multiple policies, possibly from multiple components, are violated will be confined to a single Event Analyzer. In the case where each component of the multi-tiered Web server is run on a different host, several PDPs could be configured to oversee the management of each local host where the individual component is run. However, a single Event Analyzer is used to co-ordinate the activities of the individual PDPs on each local host in order to provide quality of service support spanning multiple hosts. A key advantage of a de-centralized approach to QoS support is that the autonomic system is likely to be more scalable and responsive since QoS decisions specific to local behavior will be confined locally [16]. By reducing the distance between the *autonomous management system* and the *managed system* (see Figure 5), less overhead is incurred, in part, as a result of using more efficient local communication mechanisms between components [16]. In this section, we highlight key functionality of the different components.

4.1 Architectural Components

The following are the key components of the architecture for the adaptive policy-driven autonomic management:

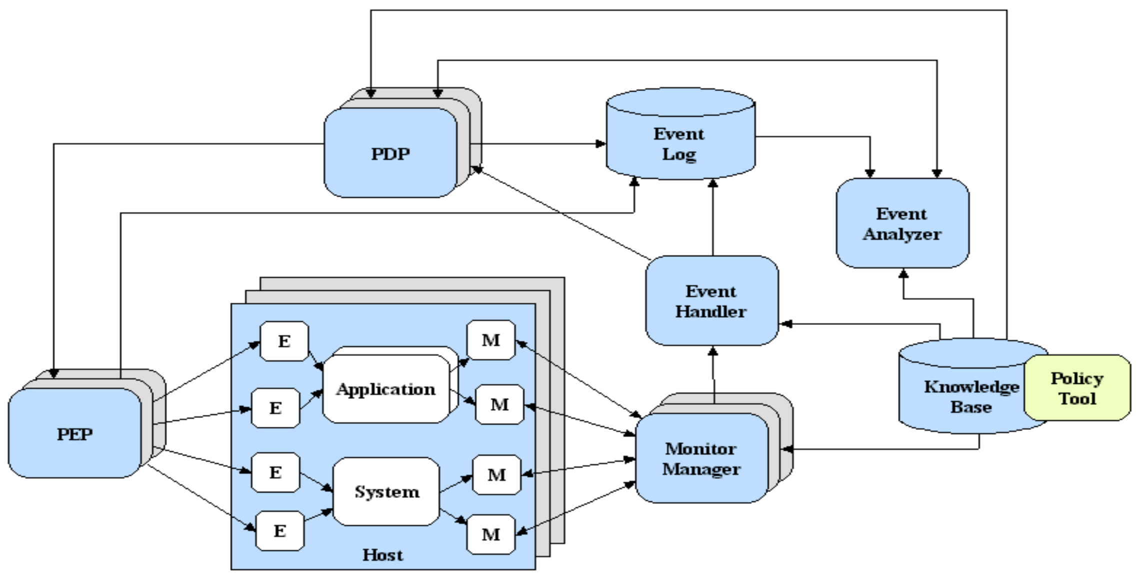


Figure 4. The adaptive policy-driven autonomic management architecture.

4.1.1 Knowledge Base

The Knowledge Base is a shared repository for system policies and other relevant information. This may include information for determining corrective actions for resolving QoS requirements violations as well as configuring systems and applications. The information about policies is eventually distributed to other management components, and then realized as actions driving the autonomic management.

4.1.2 Monitor (M)

Monitors gather performance metric information of interest for the management system such as resource utilization, response time, throughput and other relevant information. It is this information that is then used to determine whether the QoS requirements are either being met or violated.

4.1.3 Monitor Manager

Monitor Manager deals with the management of Monitors, including instantiating (i.e., loading and starting) a Monitor for a certain resource type to be monitored as well as providing the context of monitoring (i.e., monitoring frequency or time interval for periodic monitoring or monitoring times for scheduled monitoring). In addition, it allows Monitors to be re-configured (i.e., adding a new Monitor, adjusting the context of monitoring, or disabling a Monitor) dynamically in response to run-time changes to policies. At the core of its responsibility is the collection and processing of Monitor events whose details are then reported to the Event Handler. In essence, the Monitor Manager acts as an *event producer* by gathering information from multiple Monitors as illustrated in Figure 4. It provides customized services to *event consumers* (such as the Event Handler) in terms of how often they should receive events notifications.

4.1.4 Event Handler

The Event Handler deals with the processing of events from the Monitor Manager to determine whether there are any QoS requirements violations (based on the enabled policy conditions) and forwarding appropriate notifications to the interested components. This includes notifying the PDP of conditions violations as well as forwarding information to the Event Log for archiving. A key feature of this component is its ability to provide customized services to event consumers (i.e., PDP, Event Log, etc.) through subscriptions by allowing components to specify, for example, how often and/or when they should receive notifications.

4.1.5 Policy Decision Point (PDP)

This component is responsible for deciding on what actions to take given one or more violation messages from the Event Handler. The PDP must decide which policy, if any expectation policy has been violated, was the “most important” and then what action(s) to take. It uses information not only about the violations, but also the expectation policies and management policies, both expressed within the expectation policies and via management policy rules.

4.1.6 Policy Enforcement Point (PEP)

This component defines an Application Programming Interface (API) which maps the actions subscribed by the PDP to the executable elements; i.e., the various Effectors.

4.1.7 Effector (E)

Effectors translate the policy decisions, i.e., corrective actions, into adjustment of configuration parameters to implement the corrective actions. Note that there will be multiple instances of the Effectors for different types of resources (e.g., logical partitioning of CPUs, allocation of streaming buffers) or tuning parameters to be adjusted.

4.1.8 Event Log

This component archives traces of the management system's events onto (1) an event log in the memory for capturing recent short term events, and (2) a persistent event log on disk for capturing long term history events for later examination. Such events may include QoS requirements violations from the Event Handler, records of decisions made by the PDP in response to the violations, the actions enforced by the PEP, as well as other relevant management events.

4.1.9 Event Analyzer

This component correlates the events with respect to the contexts, performs trend analysis based on the statistical information, and models complex situations for causality analysis and predictive outcomes of corrective actions, to enable the PDP to learn from past, predict future and make appropriate trade-offs and optimal corrective actions.

4.2 Component Interaction

Figure 5 illustrates key interactions driving autonomic management. In this approach, we make use of policies to specify both the expected performance behavior of the managed systems as well as decisions driving autonomic management. Such policies are specified via the Policy Tool (see Figure 11). The management system can also adapt, dynamically, to handle changes to policies made via the interface. This approach is illustrated in the diagram and is characterized by the interaction between the *Managed System* and the *Autonomous Management System*. In essence, the management system determines what to monitor based on the policies that are active and determines if changes should be made. Any changes are done through effectors which can change the values of various parameters of the applications (e.g. Apache or other components) or change the operation of the system itself, such as blocking requests or adding/removing processes. This section looks at how the different components interact to achieve the different performance objectives in the context of self-configuration and self-optimization.

4.2.1 Self Configuration

Briefly, the management system in Figure 4 is instantiated by first invoking the Management Agent (not shown in the diagram). The initial task of this agent is to query all the enabled *configuration policies* (see Section 3.2.1) from the policy repository. It is these policies that are used to install the management components, with the exception of Monitors, the responsibility of which falls to the Monitor Manager; i.e., the policy `subject` (see Figure 2). The PDP, in turn, queries the policy repository for all the enabled *expectation policies* (see Section 3.2.2) and uses this information to make decisions on how to respond to violations. Once the different management components have been installed, the manager's responsibility becomes ensuring that appropriate components are notified if there are any changes to the policies governing the behavior of the system.

To illustrate the impact of disabling a policy, let's assume that the policies of Figures 3 and 9 are the only enabled expectation policies and a user disables the latter policy. This would trigger four specific notifications: (i) The first notification would be forwarded to the PDP since this component is the subject of the policy. The PDP in turn would update its policies accordingly, i.e., by removing the CPU violation expectation policy. (ii) The second notification would be forwarded to the Event Handler, the component responsible for determining whether the QoS requirements are being met. Disabling the policy of Figure 9 means that the conditions "`CPU:utilization`" and "`CPU:utilizationTREND`" must also be disabled. This would prevent any notifications from being forwarded to the PDP should a violation of any of the conditions occur. (iii) The third notification would be forwarded to the Monitor Manager to determine whether any of its Monitors are to be disabled as a result. In this particular case, the CPU Monitor would be disabled since events specific to CPU utilization are no longer relevant. This directive is captured by the test "`IsConditionEnabled(CPU:utilization)`" as part of the action to install the CPU Monitor (see the policy of Figure 2). (iv) The fourth and final notification would be forwarded to the Event Analyzer, which may need to update policy state information since the change may affect the learning process. This type of adaptation is the focus of our current research and will not be addressed here.

4.2.2 Self Optimization

Self-optimization deals with adapting the behavior of applications as well as systems in order to meet specific performance objectives. In the context of where policies are used to drive autonomic management, adaptation may be specific to the choice of policy actions. Figure 5, in particular, illustrates two main feedback control loops that drive how the autonomic system adapts the way it responds to the violations in the QoS requirements of the managed system.

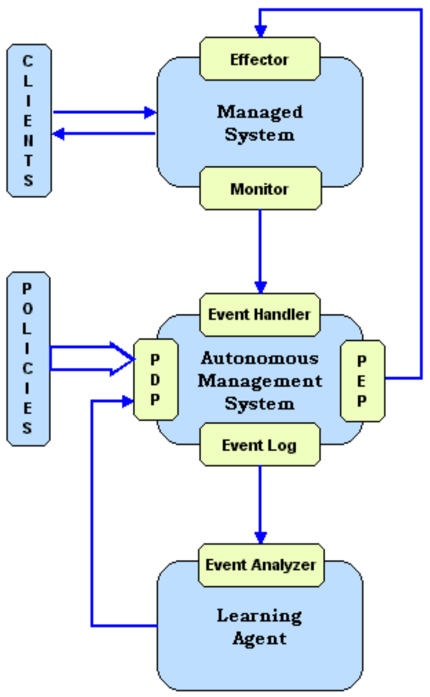


Figure 5. Two feedback loops driving the autonomic management of a managed system.

The first control loop, which constitutes a single management cycle, consists of monitoring the behavior of the managed system and, using the information collected within the interval, selecting policy actions to resolve violations.

1. The Monitors collect and forward performance metric information to the Monitor Manager (not shown in the diagram) which is then processed (i.e., for averages and trends) and forwarded to the Event Handler.
2. The Event Handler's responsibility is to determine whether the QoS requirements of the managed system have been violated. For each violation, a notification is forwarded to the PDP.
3. For each management interval, the PDP collects all the violation messages and processes them to determine whether any of the enabled expectation policies has been violated. The PDP then determines the order in which the actions advocated by the violated policies are to be "tried" (based on the violation information collected during the interval). The ordered actions are then forwarded to the PEP.
4. On receiving the policy actions, the PEP performs tests associated with each action, and if successful, invokes the appropriate Effector(s) to perform the actual adjustment to the managed system's parameter(s). Note that, in our current implementation, we only permit a single action to be executed - for the reasons discussed in Section 3.1.3.

The above control mechanisms were the focus of our initial investigation on the performance behavior of the Apache Web Server (see, for example, [17]). This work was later extended to incorporate adaptation strategies on how the PDP selects policy actions from those advocated by the violated policies in the context of a multi-component Web server (see [18]).

The second feedback loop deals with self-optimization; i.e., the ability of systems to evaluate their own behavior and adapt it accordingly to improve performance. In the context of where policies are used to drive autonomic management, this often requires monitoring the behavior in the use of policies and using the experience to learn optimal policies; i.e., the selection of optimal policy actions for each encountered situation. The use of policies in this context offers significant benefits to autonomic systems in that it allows systems administrators to focus on the specification of the objectives leaving it to systems to plan how to achieve them. The key steps of the feedback loop include the following:

1. Process the Event Log information (which includes Monitor events, QoS requirements violation events, decisions made by the PDP in response to the violations, and the actions enforced by the PEP), on-line, to model the performance of the managed system based on the observed experience in the use of policies.
2. Use the model, when possible, to advise the PDP on how to adapt its action selection mechanisms based on the current state of the system.

Figure 6 summarizes the key interactions between the different components involved in coordinating the steps of the two feedback control loops during a single management cycle. The Policy Tool (see Figure 11), in this case, provides an interface to the autonomous management system through which users can manage (i.e., add, modify, delete) policies governing the behavior of the system.

1. **Monitors:** Collect performance metric information of interest from the managed environment (E) and forward it to the Monitor Manager.

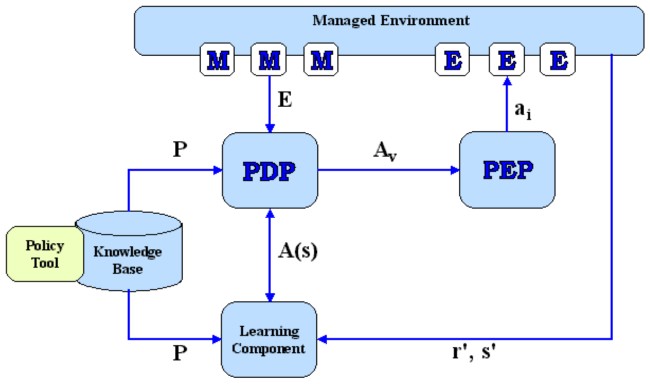


Figure 6. Feedback control interactions.

2. **Monitor Manager:** Process the Monitor events for averages and trends and forward the processed information to the Event Handler.

3. **Event Handler:** Determine whether any QoS requirements have been violated. For each violation, forward a notification, e_i, to the PDP.

4. **PDP:** During each management interval, form a set, P_v, of violated policies (from the enabled policies set P) based on the violation notification events, $e_i \in E_v$, received during the interval. A policy is said to be violated if all its conditions evaluate to true when matched against violation events in E_v.

5. **PDP:** Decide whether to use the knowledge learned from past experience with a set of active policies, P, to select the best action to take (i.e., by requesting advise from the learning component with probability $1 - \epsilon$), or to try actions not yet tried (with probability ϵ).

6. **PDP:** Form set A_v corresponding to the actions associated with the current state, $A(s)$, if the state has previously been encountered, then continue with 10 (exploit); Otherwise, continue with 7 (explore).

7. **PDP:** Compute the severity of each condition in P_v using the values of the violation events in E_v.

8. **PDP:** Form a set, A_v, of unique policy actions based on the actions advocated by the violated policies in P_v.

9. **PDP:** Compute $Q_0(s, a)$ for each policy action in A_v. $Q_0(s, a)$ estimates the initial action-value of the policy actions based on the characteristics of both violation events and the enabled policies (see Equation 6).

10. **PDP:** Sort the actions in A_v by the action-value estimate, $Q(s, a)$, and forward them to the PEP. The aim here is to ensure that actions with the highest value are tried first. Since only a single action is executed, the order in which the actions are arranged is of great importance.

11. **PEP:** Validate the policy actions in A_v by performing the tests associated with each action (see, for example, Figure 3) and then invoke the appropriate Effector (E) to perform the actual action, a_i, for the first action to pass the tests.

12. **Learning Component:** Observes the resultant state s' and reward r'. Using the Dyna-Q algorithm (see Algorithm 2), update the current system model.

In the next Section, we elaborate on how the above feedback control interactions are modelled onto a Reinforcement Learning problem.

5 Modelling Reinforcement Learning

A model, in Reinforcement Learning, describes any feedback that guides the interaction between the learning agent and its environment. This interaction is driven by the choices of actions and the behavior of the system as a consequence of taking those actions. In the context of a policy-driven autonomic management agent, the choices of actions are determined by the expectation policies that are violated. This section looks at what constitutes a state-transition model and how this structure is derived. We first begin by formally defining expectation policies.

Definition 1 *An expectation policy is defined by the tuple $p_i = \langle C, A \rangle$ where:*

- C is conjunctive conditions associated with policy p_i with each condition, $c_j \in C$, defined by the tuple $c_j = \langle \texttt{ID, metricName, operator}, \Gamma \rangle$, where; `ID` is a unique identification for the condition; `metricName` is the name of the metric associated with the condition; `operator` is the relational operator associated with the condition; and Γ is the threshold of the condition.

- A is a set of actions associated with policy p_i with each action, $a_j \in A$, defined by the tuple $a_j = \langle \texttt{ID, function, parameters}, \tau \rangle$, where; `ID` is a unique identification for the action; `function` is the name of the function (within the PEP) that should be executed; `parameters` is a set of function parameters; and τ is a set of tests associated with the action.

An expectation policy condition essentially identifies the region (or interval) on the side of the condition's threshold (based on the condition's operator) where a condition is said to be violated. Thus, our expectation policy conditions only consider "$>$, $\geq$, $<$, and $\leq$" operators. For a policy condition "`APACHE:responseTime > 2000.0`", for example, any response time measurement beyond 2000.0 ms would be considered as a violation, the severity of which increases the further the measurement is from the threshold. In our implementation of expectation policies, it is assumed that the quality of service specific to a metric's measurement deteriorates, i.e., monotonically decreases, as the measured value increases. In essence, the main objective for the autonomic manager is to steer the system towards metrics' regions where the quality of service is the highest; i.e., towards the most desirable regions.

A policy-driven autonomic management system is likely to consist of multiple expectation policies, a subset of which may be active (or enabled) at any given time; which brings us to our next definition.

Definition 2 *Suppose that P^A denotes a set of all expectation policies such that $p_i \in P^A$ where $p_i = \langle C, A \rangle$. Let P be a subset of expectation policies an autonomic manager uses to make management decisions; i.e., $P \subseteq P^A$. A policy system corresponding to P is defined by the tuple $PS = \langle P, W_C \rangle$ where:*

- $W_C = \langle c_i, \omega_i \rangle$ associates each policy condition, c_i, with a weight, ω_i, such that, for all $c_i \in p_m$ and $c_j \in p_n$, $\omega_i = \omega_j$ if $c_i = c_j$.

The conditions' weights, which are specified manually in our current implementation, provide a way of distinguishing policy conditions based on the significance of violating a particular metric. In essence, W_C provides a way of biasing how the autonomic system responds to violations; we elaborate further on this in Section 6.1.

To model system's dynamics from the use of an active set of policies, we make use of a mapping between the enabled expectation policies and the managed system's states whose structure is derived from the metrics associated with the enabled policy conditions.

Definition 3 *A policy system $PS = \langle P, W_C \rangle$ derives a set of system metrics, $m_i \in M$, such that, for each $C \in p_j$ where $p_j \in P$, $M = \bigcup_{c_i \in C} \{c_i.\texttt{metricName}\}$.*

In this approach, a *state-transition model* (see Definition 4) is defined which uses a set of active expectation policies (see, for example, Figure 3) to create a set of policy-states and the actions of the management system to determine transitions between those states. This mapping is motivated by two key observations about the expectation policies. First, they define what the expected performance and behavioral objectives are (as captured by the conditions of the enabled expectation polices). Second, they define the choices of actions whenever the specified objectives are violated. In essence, the interaction between the learning agent and its environment is driven, partly, by the enabled expectation policies. Thus, we assume a standard Markov Decision Process (MDP) [3, 4].

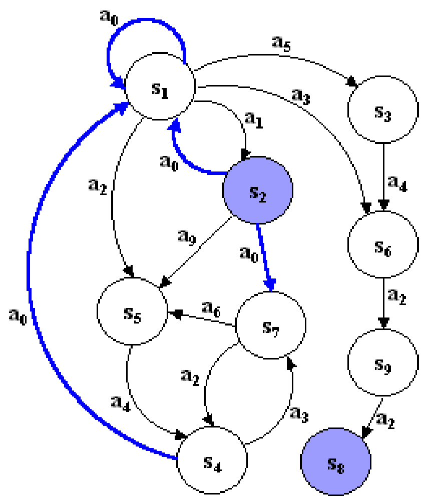


Figure 7. A sample state transition graph.

Definition 4 *A state-transition model derived from the policy system $PS = \langle P, W_C \rangle$ is defined by the graph $G^P = \langle S, T \rangle$ where:*

- S is a set of system states (see Section 5.1.) derived from the metrics of the conditions of the enabled expectation policies.
- T is a set of transitions (see Section 5.2) where each transition, $t_i \in T$, corresponds to a directed edge on the graph. A transition is determined when the autonomic manager takes an action as a result of being in one state, which may, or may not, result in a transition to another state.

As such, we capture the management system's behavior in the use of an active set of policies using a state-transition graph. This is illustrated in Figure 7 which shows the different types of states (see Definition 7) as well as transitions between states as a result of either the actions of the autonomic manager (i.e., a_i) or other dynamic characteristics outside the control of the autonomic manager (i.e., a_0); we elaborate further on this in Section 5.2. The result can then be used by the autonomic manager to consider choices of policy actions that it might take when it determines that the system is in a particular state. That is, the autonomic manager could take an action as defined below:

$$a_i \in A(s_i) \tag{1}$$

where $A(s_i)$ is a set of actions advocated by the expectation policies that are violated when the system is in state s_i. The information about the system that is used to determine the state-transition graph is extracted from the Event Log as illustrated in Figure 5. For a given set of active policies, this structure is built dynamically as the events from the different management components are recorded in the logfile. Note that, since the autonomic manager records only those states that are experienced, "states explosion" is likely to be restricted. We comment further on this in Section 9.

5.1 System States

As indicated, states are based upon the metrics in the conditions of the policy system. We define states through the following definitions.

Definition 5 *A policy system $PS = \langle P, W_C \rangle$ with metrics set M derives a set of metric-regions, M_R, for each metric $m_i \in M$, $r_{m_i} \in M_R$, whose structure is defined by the tuple $r_{m_i} = \langle \alpha_{m_i}, \sigma_{m_i} \rangle$, where:*

- $\alpha_{m_i} = \langle \texttt{ID}, \texttt{metricName}, \omega \rangle$ corresponds to a unique metric from among the metrics of the conditions of the policies in P; such that, `metricName` is the name of the metric and ω is the weight of the condition (see Definition 2) associated with metric m_i. In the case that a single metric is associated with more than one policy condition and where each condition might have different weights, the value $m_i.\omega$ is computed as follows:

  $$m_i.\omega \;=\; \max_{c.m_i \in C} c.\omega \qquad (2)$$

  which essentially corresponds to the weight of the condition with the largest weight value from among the conditions associated with metric m_i.

- $\sigma_{m_i} = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ is a set of thresholds from the conditions associated with metric m_i such that, $\Gamma_i < \Gamma_j$ if $i < j$. As such, σ_{m_i} derives a set of metric regions which map the observed metric measurement onto appropriate localities (i.e., intervals) as defined by the thresholds of the policy conditions associated with metric m_i, such that $R_{m_i} = \{R^1_{m_i}, R^2_{m_i}, \ldots, R^{k+1}_{m_i}\}$, where $R^1_{m_i} = (-\infty, \Gamma_1)$; $R^2_{m_i} = (\Gamma_1, \Gamma_2)$; and, $R^{k+1}_{m_i} = (\Gamma_k, \infty)$.

Thus, if $\sigma_{m_i} = \langle \Gamma_1, \Gamma_2 \rangle$, for example, it would yield three regions in our approach: $R^1_{m_i} = (-\infty, \Gamma_1)$, $R^2_{m_i} = (\Gamma_1, \Gamma_2)$, and $R^3_{m_i} = (\Gamma_2, \infty)$; which brings us to our next definition.

Definition 6 *Given a set of metric-regions for each metric $m_i \in M$, $r_{m_i} \in M_R$, such that $r_{m_i} = \langle \alpha_{m_i}, \sigma_{m_i} \rangle$, where σ_{m_i} derives a set of metric regions $R^j_{m_i} \in R_{m_i}$; we define a mapping function, $f(R^j_{m_i}) \rightarrow \mathbb{R}$, which assigns a numeric value to the j-th region in R_{m_i} such that, $f(R^k_{m_i}) > f(R^l_{m_i})$ if $k < l$.*

An example of such a mapping, which we make use of in our current implementation, is defined by Equation 3:

$$f(R^j_{m_i}) \;=\; 100 - (\frac{100}{n-1})(j-1) \qquad (3)$$

where n is the total number of regions in R_{m_i}. This function assigns a numeric value between 100 and 0 for each metric's region in R_{m_i}, starting from 100 for the most desirable region and decrementing at equal intervals towards the opposite end of the spectrum, whose region is assigned a value of 0. This approach guarantees that the highest value is assigned to the most desirable region (i.e., the region corresponding to the highest quality of service), assuming, of course, that the assumptions about the conditions of the expectation policies hold (see Definition 1).

Definition 7 *A policy system $PS = \langle P, W_C \rangle$ with metrics M and metrics-regions M_R derives a set of system states S such that, each state $s_i \in S$ is defined by the tuple $s_i = \langle \mu, M(s_i), A(s_i) \rangle$, and where:*

- μ is a `type` which classifies a state as either "violation" or "acceptable" depending, respectively, on whether or not there are any policy violations as a result of visiting a particular state. As noted previously, a policy is said to be violated if all its conditions evaluate to true when matched against violation notifications received during a single management cycle.

- $A(s_i)$ is a set of actions advocated by the expectation policies in P that are violated when the system is in state s_i.

- $M(s_i)$ is a set of state metrics for each metric $m_j \in M$, $r_{m_j} \in M_R$, $r_{m_j} = \langle \alpha_{m_j}, \sigma_{m_j} \rangle$, such that each state metric $s_i.m_j \in M(s_i)$ is defined as follows:

Definition 8 *A state metric $s_i.m_j \in M(s_i)$ given $\alpha_{m_j} = \langle \texttt{ID}, \texttt{metricName}, \omega \rangle$ and $\sigma_{m_j} = \langle \Gamma_1, \Gamma_2, \ldots, \Gamma_k \rangle$ is defined by the tuple $s_i.m_j = \langle \texttt{ID}, \omega, \texttt{value}, R^l_{m_j} \rangle$ where:*

- `ID` is an integer value that uniquely identify each metric $m_i \in M$.

- ω is the weight associated with metric m_i.

- `value` is the observed metric measurement, or average value when state s is visited multiple times.

m_i	c_k	Policy Condition	R_{m_i}	$R^j_{m_i}$	$f(R^j_{m_i})$
m_1			$m_1.value \leq 2000.0$	$R^1_{m_1}$	100
	c_1	`APACHE:responseTime > 2000.0`	$m_1.value > 2000.0$	$R^2_{m_1}$	0
m_2			$m_2.value \leq 0.0$	$R^1_{m_2}$	100
	c_2	`APACHE:responseTimeTREND > 0.0`	$m_2.value > 0.0$	$R^2_{m_2}$	0

Table 1. A metrics structure derived from the policy system of Example 1.

State	$R^k_{m_j}$		$f(R^k_{m_j})$		$A(s_i)$	
s_i	$R^k_{m_1}$	$R^k_{m_2}$	$f(R^k_{m_1})$	$f(R^k_{m_2})$	a_l	State action
s_1	$R^2_{m_1}$	$R^2_{m_2}$	0	0	a_0	γ-action
					a_1	AdjustMaxClients(+25)
					a_2	AdjustMaxKeepAliveRequests(-30)
					a_3	AdjustMaxBandwidth(-128)
s_2	$R^2_{m_1}$	$R^1_{m_2}$	0	100	a_0	γ-action
s_3	$R^1_{m_1}$	$R^2_{m_2}$	100	0	a_0	γ-action
s_4	$R^1_{m_1}$	$R^1_{m_2}$	100	100	a_0	γ-action

Table 2. Sample policy states based on the metrics structure of Table 1.

- $R^l_{m_j}$ is the region corresponding to a region in σ_{m_j} in which the average metric measurement (i.e., `value`) falls; i.e., if $R^l_{m_j} = (\Gamma_1, \Gamma_2)$, then $\Gamma_1 <$ `value` $< \Gamma_2$. For each such region, $f(R^l_{m_j})$ then associates a value as described by Equation 3.

Using this approach, each state can be uniquely identified by the region occupied by each state metric based on the conditions of the expectation policies and the `value` associated with each metric. That is, for a set of policies involving n metrics, each state would have n metrics $\{m_1, m_2, ..., m_n\}$ and, for each metric a specific region whose intervals are derived from the thresholds of the conditions associated with the metric. To elaborate this further, consider the following examples:

Example 1 *Suppose that, policy system $PS = \langle P, W_C \rangle$ currently consists of a single active (enabled) expectation policy shown in Figure 3 (i.e., p_1) such that $P = \{p_1\}$.*

From the conditions of the policy, states derived from the policy system of Example 1 would consist of two metrics; i.e., $M = \{m_1, m_2\}$ where m_1 ="`APACHE:responseTime`" and m_2 ="`APACHE:responseTimeTREND`". It follows from Definition 5 that $\sigma_{m_1} = \{2000.0\}$ and $\sigma_{m_2} = \{0.0\}$. As such, metric m_1 would map onto two regions; the response time is either greater than 2000.0 or not. Similarly, metric m_2 would map onto two regions; the response time trend is either greater than 0.0 or not. This is illustrated by the regions shown in Table 1. In the case of the two regions of the metric "`APACHE:responseTime`", for example, the region where the response time is "$>$ `2000.0`" would be assigned a value of 0, whereas the region where the response time is "$\leq$ `2000.0`" would be assigned a value of 100 (see Equation 3). This is because it is more desirable for the system to be in the region where the response time is not violated; i.e., "m_1.`value` $\leq$ 2000.0". Thus, given a measurement about a particular metric (i.e., m_i.`value`), Equation 3 assigns a numeric value corresponding to the appropriate metric's region where the measurement falls. It is the combination of these values over all the metrics that uniquely identify individual states.

Thus, if the policy of Figure 3 was the only policy in P, it would yield four states in our approach as illustrated in Table 2. In this case, state s_1 would be considered a "violation" state since it is the only situation which causes the policy to be violated, i.e., as a result of the violation of both policy conditions. Hence, actions set $A(s_1)$, in addition to action a_0 (i.e., do-nothing), would consist of the actions of the violated policy. The remaining three states are considered as "acceptable" states.

Example 2 *Suppose that, we extend Example 1 by adding the policy of Figure 8 (i.e., p_2) onto the policies set P such that $P = \{p_1, p_2\}$.*

It follows from Example 2 that the state metric m_1 ="`APACHE:responseTime`" would now be associated with two unique policy conditions; "`APACHE:responseTime > 2000.0`" from policy p_1 and "`APACHE:responseTime < 250.0`" from policy p_2. Consequently, the state metric "`APACHE:responseTime`" would now consist of three regions; i.e., "m_1.`value` $<$ 250.0", "250.0 $\leq$ m_1.`value` $\leq$ 2000.0", and "m_1.`value` $>$ 2000.0". In

m_i	c_k	Policy Condition	R_{m_i}	$R^j_{m_i}$	$f(R^j_{m_i})$
m_1	c_3	`APACHE:responseTime < 250.0`	$m_1.value < 250.0$	$R^1_{m_1}$	100
			$250.0 \leq m_1.value \leq 2000.0$	$R^2_{m_1}$	50
	c_1	`APACHE:responseTime > 2000.0`	$m_1.value > 2000.0$	$R^3_{m_1}$	0
m_2			$m_2.value \leq 0.0$	$R^1_{m_2}$	100
	c_2	`APACHE:responseTimeTREND > 0.0`	$m_2.value > 0.0$	$R^2_{m_2}$	0

Table 3. A metrics structure derived from the policy system of Example 2.

State	$R^k_{m_j}$		$f(R^k_{m_j})$		$A(s_i)$	
s_i	$R^k_{m_1}$	$R^k_{m_2}$	$f(R^k_{m_1})$	$f(R^k_{m_2})$	a_l	State action
s_1	$R^3_{m_1}$	$R^2_{m_2}$	0	0	a_0	γ-`action`
					a_1	`AdjustMaxClients(+25)`
					a_2	`AdjustMaxKeepAliveRequests(-30)`
					a_3	`AdjustMaxBandwidth(-128)`
s_2	$R^3_{m_1}$	$R^1_{m_2}$	0	100	a_0	γ-`action`
s_3	$R^2_{m_1}$	$R^2_{m_2}$	50	0	a_0	γ-`action`
s_4	$R^2_{m_1}$	$R^1_{m_2}$	50	100	a_0	γ-`action`
s_5	$R^1_{m_1}$	$R^2_{m_2}$	100	0	a_0	γ-`action`
					a_4	`AdjustMaxClients(-25)`
					a_5	`AdjustMaxKeepAliveRequests(+30)`
					a_6	`AdjustMaxBandwidth(+64)`
s_6	$R^1_{m_1}$	$R^1_{m_2}$	100	100	a_0	γ-`action`
					a_4	`AdjustMaxClients(-25)`
					a_5	`AdjustMaxKeepAliveRequests(+30)`
					a_6	`AdjustMaxBandwidth(+64)`

Table 4. Sample policy states based on the metrics structure of Table 3.

this case, the values assigned by Equation 3 to the above three regions would be 100, 50, and 0, respectively, as shown in Table 3. As a result, the policy system of Example 2 would yield six states in our approach as illustrated in Table 4 where states s_1, s_5, and s_6 would be considered as "violation" states whereas the remaining states would be considered as "acceptable" states. Thus, depending on the number of active policies in a set as well as the number of different metrics and different conditions on those metrics, the number of potential policy-states *could* be quite large; we comment further on this in Section 9. A key distinction between this and other related work (see, for example, [19, 20, 21, 22, 23]) is that the state structure is dependent only on the enabled expectation policies and can thus be automatically determined once a set of policies is specified.

5.2 System Transitions

Transitions are essentially determined by the actions taken by the management system and labelled by a value determined by our Reinforcement Learning algorithm. Which brings us to the next definition:

Definition 9 *Let $G^P = \langle S, T\rangle$ be a state transition graph for the policy system $PS = \langle P, W_C\rangle$ such that $t_i(s_p, a_p, s_c) \in T$. A state transition $t_i(s_p, a_p, s_c)$ is a directed edge corresponding to a transition originating from state s_p and ending on state s_c as a result of taking action a_p while in state s_p, and is labelled by $\langle\lambda, Q_{t_i}(s_p, a_p)\rangle$, where:*

- λ is the frequency (i.e., the number of times) through which the transition occurs.
- $Q_{t_i}(s_p, a_p)$ is the action-value estimate associated with taking action a_p in state s_p. In our current implementation, $Q_{t_i}(s_p, a_p)$ is computed using a one-step Q-Learning [3] algorithm (see Equation 4).

A change in the system's state may also be due to external factors other than the impact of the actions taken by the autonomic manager. In a dynamic Web server environment, for example, a transition may be a result of a request to a page with a database-intensive query, which could potentially cause a state transition. These are modeled in the state-transition graphs as γ-transitions; the actions responsible for such transitions are denoted by a_0 (i.e., γ-`action`) as illustrated in Table 4.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (4)$$

```
expectation policy{RESPONSETIMENormal(PDP, PEP)}
 if(APACHE:responseTime < 250.0)
 then{AdjustMaxClients(−25) test{newMaxClients > 49} |
      AdjustMaxKeepAliveRequests(+30) test{newMaxKeepAliveRequests < 91} |
      AdjustMaxBandwidth(+64) test{newMaxBandwidth < 1281}}
```

Figure 8. An expectation policy for dealing with an improvement in the server's response.

5.3 Reward Function

The main objective (or *goal*) of the autonomic manager, in essence, is to learn an optimal policy for "steering" the system towards "acceptable" states and away from "violation" states. In order to achieve this, a numeric reward, r, must be defined after each time step during which the agent acts (see, for example, Figure 1) to indicate the desirability of taking a particular action in a given situation (i.e., state s_t). What, then, should the reward be in order to encourage the learning of optimal behavior?

We are currently exploring one approach for deriving the reward signal, such that the learning agent is encouraged to take actions which may eventually lead to "acceptable" states. Rather than taking the simplest approach whereby an agent is only rewarded if an action results in a transition to such a state, we associate each state (both "violation" and "acceptable") with a reward value (derived from the state's metrics), which measures the desirability of the agent being in a particular state. We take this approach for three main reasons:

1. We do not make any assumptions about the accuracy of the enabled expectation policies since our main objective is to evaluate the effectiveness of these policies and, if necessary, adapt their use accordingly in order to meet specific objectives. We cannot assume, for example, that the use of an active set of expectation policies as is would be sufficient to effectively resolve the violations in QoS requirements; i.e., completely steer the system from "violation" to "acceptable" behavior.

2. The main objective of the learning agent is to figure out how to effectively use existing policies. Thus, while it may not always be possible to achieve the final objective based on the current set of active policies, the agent could still learn how at least to steer the system "towards" acceptable behavior. For example, if the objective (as defined by the enabled expectation policies) is to ensure that violations in the server's CPU and memory utilization are resolved, then we would consider a state where only a single metric is violated as better, i.e., closer to the acceptable behavior than, say, a state where both metrics are violated[1]. We could even go a step further by also considering the significance of state metrics. It could be that a violation in CPU utilization carries more weight than, say, that of memory utilization. Thus, the agent could be rewarded more generously for taking actions which result in no violation in CPU utilization, but less generously if those actions lead to no violations in memory utilization. We elaborate further on this in the next section.

3. The dynamicity of the system in terms of the changes in the state structure as a result of run-time policy modifications necessitates more flexibility in terms of how the reward function is derived. This is the focus of our current research on adaptation strategies and is beyond the scope of this paper.

Thus, we associate each state with a reward whose value increases towards acceptable behavior. From the example above, a reward is zero if the action leads to a state where both CPU and memory utilization are violated (since $f(R^j_{m_i})$ is 0 for both metrics), and is the highest for a state with no violation (since $f(R^j_{m_i})$ is 100 for both metrics). And this brings us to our next definition.

Definition 10 *Given the current system state $s_t = \langle \mu, M(s_t), A(s_t) \rangle$, such that $m_i \in M(s_t)$; an agent visiting state s_t after taking action a in the previous state is rewarded as follows:*

$$r(s_t) = \sqrt{\sum_{i=1}^{n} m_i.\omega \times [f(R^j_{m_i})]^2} \qquad (5)$$

where, n is the number of metrics, and $m_i.\omega$ and $R^j_{m_i}$ correspond, respectively, to the weight associated with metric m_i and the region where metric m_i measurement falls (see Definition 8). In essence, Equation 5 assigns each state a reward whose value increases as one moves towards the most desirable states.

[1]In this example, there would be four states since each state metric could have two possible regions; either it is violated or not. Thus, the following states are possible; (i) a state where both metrics are violated, (ii) a state where only CPU utilization is violated, (iii) a state where only memory utilization is violated, and (iv) a state where neither metric is violated.

6 Learning by Reinforcement

Central to the functionality of the PDP is the need to determine what actions to take given certain violations in QoS requirements. Note that the choice of actions, $a \in A(s)$, is dependent on the expectation policies that are violated when the system is in state s. Since policy violations are triggered by Monitor events collected during the current management interval, the PDP must decide whether to base its action selection decisions on this information alone or whether to request advice based on past experience when making those decisions. This decision-making dilemma lends itself well to the *explore-exploit* dilemma in Reinforcement Learning and is explored in detail next. But first, we comment briefly on key characteristics that could influence the choice of the algorithm for balancing exploration and exploitation.

- Each action $a \in A(s)$ cannot be treated equally, particularly because of the importance of the order in which the actions are specified within each expectation policy. As illustrated by the the expectation policy of Figure 3, it is often the case that more drastic actions (i.e., `AdjustMaxBandwidth(-128)` which throttles clients requests by reducing server's network bandwidth) are taken once it is no longer possible, for example, to meet the specified objectives through the adjustment of applications tuning parameters. It is therefore important that, to some extent, this order is preserved, at least during the initial phase of the learning process.

- It is often the case that characteristics specific to the violations (and not just the type of violation) provide useful information about the state of the system as well as how to best respond to the situation. For example, if the aim is to ensure that the server's response does not exceed 2000.0 ms, then it might be desirable to treat a violation in the server's response time of 5000 ms differently than, say, a violation of 2001 ms. Such kind of information could also be useful in guiding the exploration process to ensure that more urgent needs are addressed first.

- We note also that exploration could be quite costly especially in situations where excessive penalties are incurred. It is, therefore, important that the exploration process takes advantage of existing knowledge about the policies and violation events as opposed to selecting policy actions based exclusively on the type of violations.

To this end, we propose the use of a *near greedy* approach to balancing exploration and exploitation whereby the learning agent behaves greedily - by executing the action with the highest $Q(s, a)$ - most of the time (with probability $1 - \epsilon$) and, once in a while (with probability ϵ) the agent selects an action independent of the current action-value estimate $Q(s, a)$. Unlike the ϵ-*greedy* method [3] which treats all actions equally during exploration, action selection is based on the action-value estimate that is derived from the characteristics of both policies and violations. This is particularly useful when it is necessary to differentiate one action from another given that multiple, and at times conflicting, actions may be "advocated" by the violated policies and where the order in which the actions are specified might be of importance.

6.1 Exploration Strategy

In certain situations, the learning agent may need to make management decisions without depending, exclusively, on past experience. This could be part of the agent's strategy of exploring its environment to discover what actions bring the most reward. It could also be because the agent may have no other choice if past experience does not include knowledge about the current situation if, in fact, it is the first time the situation is encountered. Consequently, the agent may have to base its decisions on information other than past experience. In our approach, these decisions are guided by the following strategies that are based on the characteristics of the enabled expectation policies and those of the violation events:

1. *The severity of the violation:* Rather than treating each violation equally, we assign more weight to those violations that are more severe. The severity of the violation is based on the value of the metric relative to the condition's threshold. For example, for a CPU utilization of 100% given the condition "`CPU:utilization > 85.0`" (i.e., as a result of violating the policy of Figure 9), this value is computed from the difference between the measured value and its threshold value (i.e., 15%) as defined by Equation 8.

2. *The significance of the violation:* In the case that multiple policies are violated, it may be desirable to assign a higher priority (or weight) to a particular event so that the management system can respond to such a violation (i.e., by selecting appropriate policy actions) first before dealing with other less-important violations. For instance, it is quite reasonable to respond to CPU utilization violations before addressing violations related to, say, response time since failure to address the former may result in more severe violations of the latter as a result of over-utilization of CPU resources. This is done by allowing a weight to be associated with events which then become weights on the conditions that become true in violated policies (see Definition 2). The weight associated with policy con-

```
expectation policy{CPUViolation(PDP, PEP)}
 if(CPU:utilization > 85.0) & (CPU:utilizationTREND > 0.0)
 then{AdjustMaxClients(-25) test{newMaxClients > 49} |
      AdjustMaxKeepAliveRequests(-30) test{newMaxKeepAliveRequests > 1} |
      AdjustMaxBandwidth(-128) test{newMaxBandwidth > 255}}
```

Figure 9. An expectation policy for resolving Apache's CPU utilization violation.

```
expectation policy{CPUandRESPONSETIMEViolation(PDP, PEP)}
 if(CPU:utilization > 85.0) & (CPU:utilizationTREND > 0.0) &
   (APACHE:responseTime > 2000.0) & (APACHE:responseTimeTREND > 0.0)
 then{AdjustMaxKeepAliveRequests(-30) test{newMaxKeepAliveRequests > 1} |
      AdjustMaxBandwidth(-128) test{newMaxBandwidth > 255}}
```

Figure 10. An expectation policy for resolving Apache's CPU utilization and response time violations.

dition c_i which then becomes the strength of policy p_j is denoted by the parameter $c_i.\omega$ (see Equation 7).

3. *The advocacy of the action:* In the case that multiple policies are violated, it might be possible that more than one policy advocates the same action. For example, in our current test environment involving the Apache server and other components, different policies with different conditions (see, for example, Figures 3 and 9) may indicate that the same action be taken, i.e., `AdjustMaxBandwidth` which controls the maximum number of requests a server can process. The number of policies advocating the action as well as the position of the action within each policy (whose weight is denoted by the parameter $W_a(p_j)$ in Equation 6) are also considered when estimating $Q_0(s,a)$. The position is of particular interest since, in our experience, it is often the case that more drastic actions are not taken until other actions to adjust tuning parameters have first been "tried".

4. *The specificity of the policy:* In a situation where several policies are violated, the number of conditions within each policy (as well as conditions weights) could also be taken into consideration when determining which policy has more weight. For example, in the event that both CPU utilization and response time are violated, the policy in Figure 10 would be given more weight than the policy of Figure 9. This information could be taken into account when evaluating the strength of policy p_j, which we refer to as $S(p_j)$ (see Equation 7).

Thus, given the policy system $PS = \langle P, W_C \rangle$ (see Definition 2) and supposing that P_v is a set of expectation policies that are violated in the current management interval such that $P_v \subseteq P$, we can estimate the initial value of an action, a, as follows:

$$Q_0(s,a) = \frac{\sum_{p_j \in [P_v]_a} tanh[S(p_j)] \times W_a(p_j)}{\parallel [P_v]_a \parallel} \tag{6}$$

where $[P_v]_a$ is the subset of violated policies advocating action a; $W_a(p_j)$ is the weight of action a based on its position within policy p_j. In our current implementation, actions weights take values between 100 and 0 such that the first policy action gets the highest value (i.e., 100) while the last policy action gets the lowest value (i.e., 0), with weights assigned to the actions at equal intervals according to Equation 3. Thus, in the case of a policy with three actions such as the policy of Figure 9, the values would be 100, 50, and 0, in that order; $S(p_j)$ is the strength of policy p_j as specified by Equation 7:

$$S(p_j) = \sum_{c_i \in p_j} c_i.\omega \times V(c_i) \tag{7}$$

where $c_i.\omega$ is the weight associated with policy condition c_i based on the significance of the condition's violation (see Definition 2), and $V(c_i)$ is the severity of the violation of condition c_i. This value is computed as follows:

$$V(c_i) = \left| \frac{e_i.\texttt{value} - c_i.\Gamma}{\Omega} \right| \tag{8}$$

where e_i.`value` is the current value of the event responsible for violating condition c_i, $c_i.\Gamma$ is the threshold value of condition c_i, and

$$\Omega = \begin{cases} 1, & |c_i.\Gamma| \le 1 \\ c_i.\Gamma, & otherwise \end{cases} \tag{9}$$

Briefly, Equation 7 estimates $Q_0(s, a)$ based on the *severity* of the violations of the conditions associated with, as well as the *significance* of, individual policies. We measure the severity based on the difference between the condition's threshold, $c_i.\Gamma$, and the observed value of the metrics, e_i.value, responsible for its violation (see Equation 8). In certain situations, it may be desirable to designate a higher priority to a particular event so that the management system can respond to such a violation first (i.e., by selecting appropriate policy actions) before dealing with other less important violations. This is the purpose of the parameter $c_i.\omega$ in Equation 7. This information could then be used to estimate the action value (see Equation 6), which takes into account the number of policies advocating the action, the position of the action, and the severity associated with the violation of the conditions within each violated policy. Thus, the same set of violations, for example, may result in different actions being taken depending on the initial action-value estimates. This is in contrast to static approaches where the order of the actions is always the same for the same set of violations.

6.2 Exploitation Strategy

As noted previously, it is often very difficult to obtain, in advance, models that accurately capture systems dynamics particularly for the state of the enterprise systems. Our approach to learning, for reasons mentioned in Section 2, is based upon the Dyna-Q framework [8] where the model of the system is continuously learned, on-line, and used for planning. We are currently exploring several strategies on how such a model, represented by the state-transition graph (as discussed in Section 5), might be used to help the system adapt the way it uses policies when making decisions on how to resolve QoS requirements violations. These strategies fall into two broad categories:

1. **Reactive Enforcement**: In this approach, the autonomic manager could adapt the way it reacts to violations in QoS requirements (i.e., respond after a violation has occurred) based on the currently learned model. One such approach involves having the PDP request advice from the learning component during each management cycle where the system is in "violation" state. This may include, for example, an advice on what policy action to take in the current state (i.e., s) based on the currently learned $Q(s, a)$ estimates associated with each action $a \in A(s)$. It may also be possible to recommend multiple actions if their impact is deemed positive. This may involve, for example, computing the shortest path from the current "violation" state to an "acceptable" state based on the $Q(s, a)$ estimates associated with the actions within the current state-transition graph. A path, in this case, constitutes an ordered list of actions. For instance, if the system is in state s_1 of Figure 7, the learning agent may recommend the enforcement of a set of actions consisting of $\{a_3, a_2, a_2\}$ essentially steering the system to an "acceptable" state s_8.

2. **Proactive Enforcement**: In this approach, the autonomic manager, in anticipating possible violations in QoS requirements, may recommend a set of actions aimed at steering the system away from "violation" states before the system gets there. For instance, if it has been observed that the system makes a γ-transition from an "acceptable" state (i.e., s_2 in Figure 7) to a "violation" state (i.e., s_1) with a very high probability, then appropriate actions could be taken before the system gets to state s_1. Thus, actions $\{a_3, a_2, a_2\}$ could be enforced while the system is still in state s_2 which may, as a result, move the system to a more stable "acceptable" state (i.e., s_8) consequently minimizing possible future violations.

The above two approaches to QoS provisioning highlight several key advantages on how the autonomic management system can respond to violations: First, rather than restricting the selection of policy actions to only those advocated by the violated policies (i.e., $a \in A(s)$), the autonomic manager is able to look beyond the actions within a single state for actions, some of which might not even be part of those in the violated policies, whose impact may be positive but not immediate. Second, the autonomic manager could take multiple actions. Assume, for example, that the system is in state s_1 and that a_2 corresponds to the action "`AdjustMaxClients(+25)`" as specified by the policy of Figure 3. Thus, instead of increasing `MaxClients` by 25, the same action could be performed twice. A key advantage here is that multiple adjustments to the tuning parameters could be made when past behavior suggests that it is likely prudent to do so. Third, the autonomic manager has the ability to be proactive, that is, use past experience to take actions in anticipation of policy violations. This would be done by looking ahead in the state graph. The agent may determine whether some action could lead to either a very bad situation or a very good one. For instance, the agent using the state-transition information in Figure 7 could avoid actions such as a_2 while in state s_1 if past experience show that, once that action is taken, it is less likely for the system to make a transition back to an acceptable state.

6.3 The Learning Algorithm

To compute the action-value estimates, we use a modified version of the Dyna-Q algorithm (see Algorithm 1) that enables the agent to learn in non-deterministic environments. The algorithm (see Algorithm 2), which we refer to herein as Dyna-Q*, takes into account transition probabilities when computing action-value estimates.

Algorithm 2 Dyna-Q* Algorithm

```
Input: Initialize $G^P = \langle S, T \rangle$ for policy system $PS = \langle P, W_C \rangle$
 1: for $i = 1\ to\ \infty$ do
 2:     $s \leftarrow$ current (non terminal) state
 3:     $a \leftarrow \epsilon$-greedy$(s, Q_0)$ (see Equation 6)
 4:     Execute $a$; observe resultant state, $s'$
 5:     $Q(s,a) \leftarrow Q(s,a) + \alpha\{E[r(s,a)] + \gamma E[\max_{a'} Q(s',a')] - Q(s,a)\}$
 6:     $G^P \leftarrow \langle s', t(s,a,s') \rangle$
 7:     for $j = 1\ to\ k$ do
 8:         $s \leftarrow$ random previously observed state
 9:         $a \leftarrow$ random action previously taken in $s$
10:         $Q(s,a) \leftarrow Q(s,a) + \alpha\{E[r(s,a)] + \gamma E[\max_{a'} Q(s',a')] - Q(s,a)\}$
11:     end for
12: end for
```

To compute the expected values, we define a transition probability as follows:

$$Pr[t(s,a,s')] = \frac{t_i(s,a,s').\lambda}{\sum\limits_{t_i(s,a,s'_j) \in T(s)} t_i(s,a,s'_j).\lambda} \qquad (10)$$

where $t_i(s,a,s').\lambda$ is the frequency associated with a transition originating from state s and terminating at state s' as a result of taking action a in state s (see Definition 9). Thus, $t_i(s,a,s'_j) \in T(s)$ is a subset of transitions originating from s (i.e., $T(s)$) as a result of taking action a. From Equation 10, the expected reward can be computed as follows:

$$E[r(s,a)] = \sum_{t_i(s,a,s'_j) \in T(s)} Pr[t_i(s,a,s'_j)] \times r(s'_j) \qquad (11)$$

where $r(s'_j)$ is the reward associated with state s'_j computed using Equation 5. Similarly, the expected action-value estimate can be computed as follows:

$$E[\max_{a'} Q(s',a')] = \sum_{t_i(s,a,s'_j) \in T(s)} Pr[t_i(s,a,s'_j)] \times \max_{a'} Q(s'_j,a') \qquad (12)$$

Note that, in the case of deterministic transitions, $Pr[t_i(s,a,s')] = 1$. Thus, $E[r(s,a)]$ is essentially equal to $r(s'_j)$; i.e., the reward the agent receives after making a transition to state s'_j (see Definition 10). Similarly, $Q(s,a)$ is essentially the same as the action-value estimate associated with the transition (i.e., $Q_{t_i}(s,a)$) as computed by Equation 4. And this is consistent with the implementation of the Dyna-Q Algorithm in deterministic environments as described in Section 2 (see Algorithm 1).

7 Results and Experience

This section presents the prototype implementation of the adaptive policy-driven autonomic system as well as report on our experience.

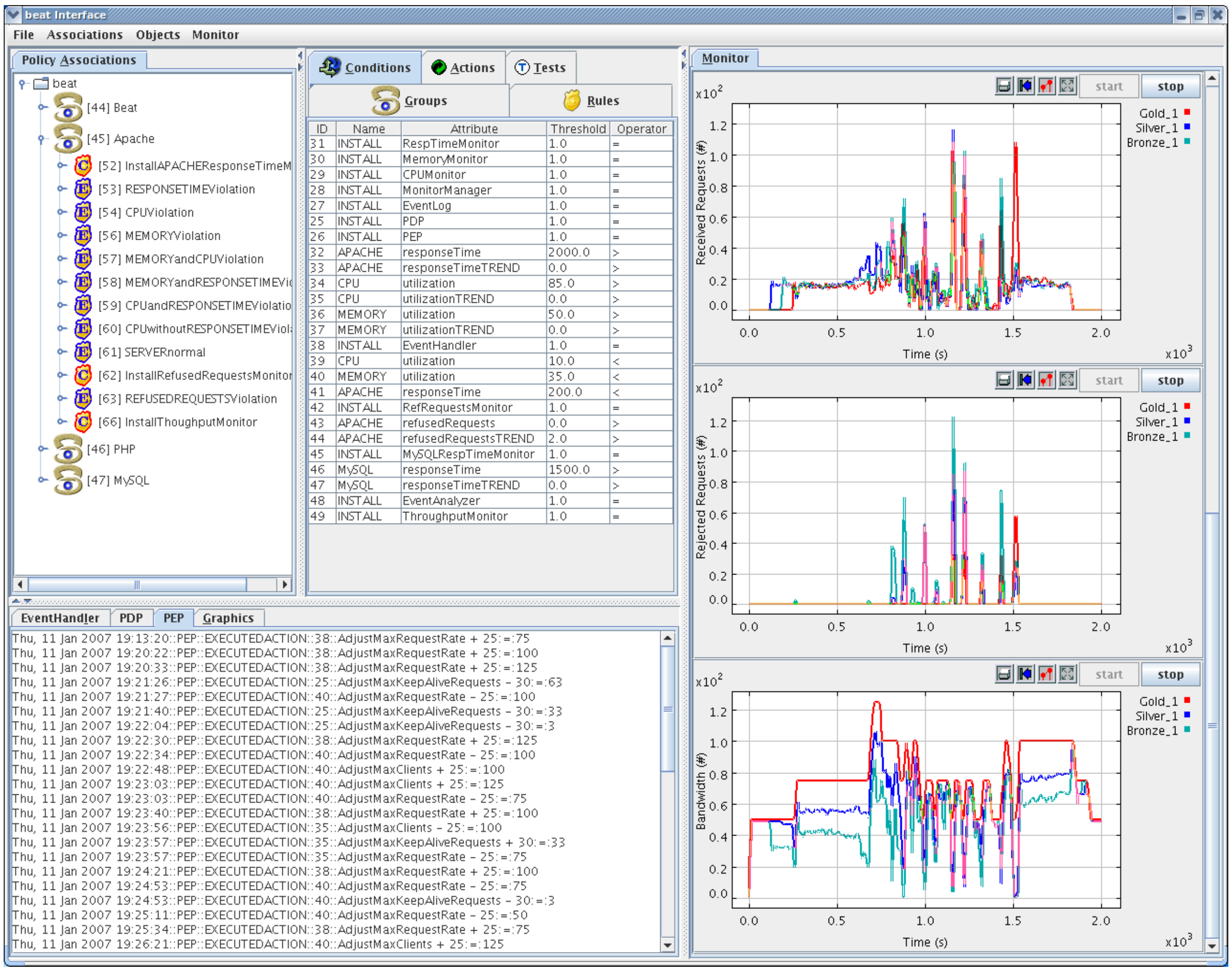
7.1 Managed System

We evaluated the effectiveness of the learning mechanisms on the behavior of a multi-component Web server consisting of an Apache (v2.2.0) [24] which was configured with a PHP (v5.1.4) module [25], and a MySQL (v5.0) database server [26]. We used the PHP Bulletin Board (phpBB) application [27] to generate dynamic Web pages. This application utilizes queries to display information stored inside a database, in our case, the MySQL database. The main database tables include forums, topics, posts, users, and groups. These tables are used to store information specific to discussions. In addition to viewing forum-related information, users may post messages using forms, which can be viewed through a Web browser. A single workstation was used to host the components as illustrated in Figure 12. Service differentiation mechanisms for classifying gold, silver, and bronze clients were also implemented (see Section 7.3). Several effectors were implemented and included those for adjusting the following parameters: (For a detailed description of the tuning parameters excluding `MaxBandwidth`, the reader is referred to [24, 26].)

- `MaxClients` (Apache): controls the maximum number of server processes that may exist at any one time (i.e., the size of the worker pool) and corresponds to the number of simultaneous connections that can be serviced. Setting this value too low may result in new connections being denied. Setting it too high, on the other hand, allows multiple clients' requests to be processed, but may lead to performance degradation as a result of excessive resource utilization.

- `MaxKeepAliveRequests` (Apache): corresponds to the maximum number of requests that a *keep-alive* connection [28] can transmit before it is closed. Its value is often set relative to the `KeepAliveTimeout`, which corresponds to the client's think time - the amount of time, in seconds, the server will wait on a persistent connection before closing it. Setting this value too high may result in having connections linger for too long after a client has disconnected thus wasting server's resources. On the other hand, setting this value too low may lead to having clients rebuild their connections often, possibly impacting the response time and CPU utilization.

- `EaccMemSize` (PHP): The PHP performance was further enhanced with the eAccelerator [29] encoder.



ID	Name	Attribute	Threshold	Operator
31	INSTALL	RespTimeMonitor	1.0	=
30	INSTALL	MemoryMonitor	1.0	=
29	INSTALL	CPUMonitor	1.0	=
28	INSTALL	MonitorManager	1.0	=
27	INSTALL	EventLog	1.0	=
25	INSTALL	PDP	1.0	=
26	INSTALL	PEP	1.0	=
32	APACHE	responseTime	2000.0	>
33	APACHE	responseTimeTREND	0.0	>
34	CPU	utilization	85.0	>
35	CPU	utilizationTREND	0.0	>
36	MEMORY	utilization	50.0	>
37	MEMORY	utilizationTREND	0.0	>
38	INSTALL	EventHandler	1.0	=
39	CPU	utilization	10.0	<
40	MEMORY	utilization	35.0	<
41	APACHE	responseTime	200.0	<
42	INSTALL	RefRequestsMonitor	1.0	=
43	APACHE	refusedRequests	0.0	>
44	APACHE	refusedRequestsTREND	2.0	>
45	INSTALL	MySQLRespTimeMonitor	1.0	=
46	MySQL	responseTime	1500.0	>
47	MySQL	responseTimeTREND	0.0	>
48	INSTALL	EventAnalyzer	1.0	=
49	INSTALL	ThroughputMonitor	1.0	=

```
Thu, 11 Jan 2007 19:13:20::PEP::EXECUTEDACTION::38::AdjustMaxRequestRate + 25:=:75
Thu, 11 Jan 2007 19:20:22::PEP::EXECUTEDACTION::38::AdjustMaxRequestRate + 25:=:100
Thu, 11 Jan 2007 19:20:33::PEP::EXECUTEDACTION::38::AdjustMaxRequestRate + 25:=:125
Thu, 11 Jan 2007 19:21:26::PEP::EXECUTEDACTION::25::AdjustMaxKeepAliveRequests - 30:=:63
Thu, 11 Jan 2007 19:21:27::PEP::EXECUTEDACTION::40::AdjustMaxRequestRate - 25:=:100
Thu, 11 Jan 2007 19:21:40::PEP::EXECUTEDACTION::25::AdjustMaxKeepAliveRequests - 30:=:33
Thu, 11 Jan 2007 19:22:04::PEP::EXECUTEDACTION::25::AdjustMaxKeepAliveRequests - 30:=:3
Thu, 11 Jan 2007 19:22:30::PEP::EXECUTEDACTION::38::AdjustMaxRequestRate + 25:=:125
Thu, 11 Jan 2007 19:22:34::PEP::EXECUTEDACTION::40::AdjustMaxRequestRate - 25:=:100
Thu, 11 Jan 2007 19:22:48::PEP::EXECUTEDACTION::40::AdjustMaxClients + 25:=:100
Thu, 11 Jan 2007 19:23:03::PEP::EXECUTEDACTION::40::AdjustMaxClients + 25:=:125
Thu, 11 Jan 2007 19:23:03::PEP::EXECUTEDACTION::40::AdjustMaxRequestRate - 25:=:75
Thu, 11 Jan 2007 19:23:40::PEP::EXECUTEDACTION::38::AdjustMaxRequestRate + 25:=:100
Thu, 11 Jan 2007 19:23:56::PEP::EXECUTEDACTION::35::AdjustMaxClients - 25:=:100
Thu, 11 Jan 2007 19:23:57::PEP::EXECUTEDACTION::35::AdjustMaxKeepAliveRequests + 30:=:33
Thu, 11 Jan 2007 19:23:57::PEP::EXECUTEDACTION::35::AdjustMaxRequestRate - 25:=:75
Thu, 11 Jan 2007 19:24:21::PEP::EXECUTEDACTION::38::AdjustMaxRequestRate + 25:=:100
Thu, 11 Jan 2007 19:24:53::PEP::EXECUTEDACTION::40::AdjustMaxRequestRate - 25:=:75
Thu, 11 Jan 2007 19:24:53::PEP::EXECUTEDACTION::40::AdjustMaxKeepAliveRequests - 30:=:3
Thu, 11 Jan 2007 19:25:11::PEP::EXECUTEDACTION::40::AdjustMaxRequestRate - 25:=:50
Thu, 11 Jan 2007 19:25:34::PEP::EXECUTEDACTION::38::AdjustMaxRequestRate + 25:=:75
Thu, 11 Jan 2007 19:26:21::PEP::EXECUTEDACTION::40::AdjustMaxClients + 25:=:125
```

Figure 11. A Graphical User Interface (GUI) to the autonomous management system.

This module provides mechanisms for caching compiled scripts so that later requests invoking similar scripts do not incur compilation penalty. The parameter `EaccMemSize` controls the size of memory cache.

- `KeyBufferSize` (MySQL): corresponds to the total amount of physical memory used to index database tables.
- `ThreadCacheSize` (MySQL): corresponds to the number of threads the database server may cache for reuse. Thus, instead of creating a new thread for each request to the database, the server uses the available threads in the cache to satisfy the request. This has the advantage of improving the response time as well as the CPU utilization.
- `QueryCacheSize` (MySQL): corresponds to the maximum amount of physical memory used to cache query results. Thus, a similar query to previously cached results will be serviced from memory and not from disk.
- `MaxConnections` (MySQL): corresponds to the maximum number of simultaneous connections to the database.
- `MaxBandwidth` (System): corresponds to the physical capacity (in kbps) of the network connection to the workstation hosting the servers.

The servers provide support for dynamic adjustment of the parameters. For the Apache-PHP server, for example, the actual adjustment to the parameters was done by editing the appropriate configuration file and performing a *graceful restart* [24] of the server.

7.2 Using Policies

We used the Policy Tool of Figure 11 to specify policies which expressed the desired behavior of the managed system (in terms of CPU, memory utilization, and response time thresholds) as well as possible management actions to be taken whenever those objectives were violated (see, for example, the policy of Figure 3). We also defined several policies that dealt specifically with the optimization of resource usage whenever an opportunity arose. This is illustrated by the policy of Figure 8 where, given that there are no violations in QoS requirements, one might reduce the number of `MaxClients` to a smaller value, thus reducing memory utilization. During this time, existing clients might also be allowed to hold onto server processes for much longer (i.e., by increasing `MaxKeepAliveRequests`) to improve their response time (rather than requiring them to re-negotiate their connections every so often). One might also increase the server's bandwidth. In our implementation, we classify states associated with the violations of such policies as "acceptable" (see Definition 7). Each state consisted of ten metrics corresponding to equally weighted (see Definition 2) conditions from the enabled policies.

7.3 Testbed Environment

A testbed environment consisted of a collection of networked workstations, each connected via 10/100 megabit-per-second (Mbps) Ethernet switch (see Figure 12). They include an administrative console used to run the Policy Tool; a Linux workstation with a 2.0 GHz processor and 2.0 Gigabytes of memory which hosted the Apache Web Server along with the Knowledge Base and the MySQL database server; and three workstations used to run the traffic load tool for generating server requests for the gold, silver and bronze service classes.

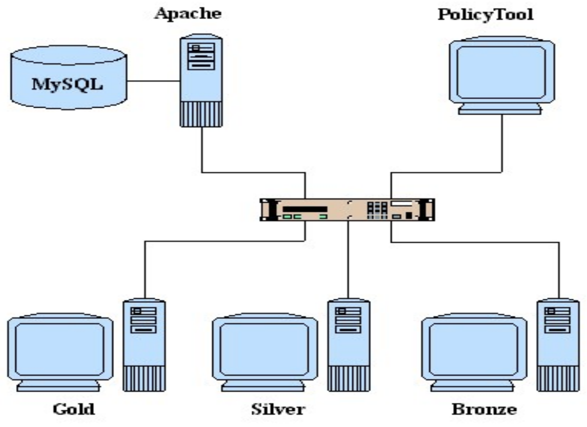


Figure 12. Testbed Environment.

In order to support service differentiation, a Linux Traffic Controller (TC) Tool [30] was used to configure the bandwidth associated with the gold, silver, and bronze service classes. Thus, given the maximum possible bandwidth the service classes throughput were assigned proportionately according to the ratio 85:10:5; bandwidth sharing was also permitted. The actual classification was based on the remote IP address of the clients' request and occurred at the point where requests reached the workstation hosting the Apache server. The tuning parameter `MaxBandwidth` is what determines how much bandwidth is assigned to each service class. Thus, given that the policy of Figure 3 has been violated and that it is no longer possible, for example, to adjust the parameters `MaxClients` and `MaxKeepAliveRequests`, then the last policy action (i.e., `AdjustMaxBandwidth(-128)`) would be executed, which essentially reduces the total bandwidth by 128 kbps. The percentage of the new bandwidth is what is eventually assigned to the different service classes.

7.4 Workload Generator

To simulate the stochastic behavior of users, the Apache load generator tool (ab) [24] was modified to support concurrent and independent keep-alive requests to the server. The tool was also modified to emulate the actual behavior of users by traversing the Web graph of an actual Web site. Thus, for each response from the server, the tool randomly selects which subsequent link (among the links in the received Web page) to follow. In the experiments reported in this paper, we only considered requests involving dynamic Web content through the use of the phpBB application. Also, we only considered database *read-only* requests. For all the experiments, the load generator in each client's workstation was configured such that the number of concurrent connections to the server and the think-time for the gold, silver, and bronze clients were identical. These values were set to ensure that the server was under overload conditions (i.e., saturated) for the duration of the experiment.

7.5 Experiments and Results

To evaluate the impact of the learning mechanisms on the behavior of the server - which was measured in terms of Apache's responsiveness (i.e., response time), throughput (i.e., number of requests processed), and resources utilization (i.e., CPU and memory) - we conducted three experiments: The first (base) experiment (*Exp-1*) looked at the behavior when all the expectation policies were disabled. The server's bandwidth was also set arbitrarily large and service differentiation mechanisms were disabled. The second experiment (*Exp-2*) looked at the impact of the action selection mechanisms (see Equation 6) which depended ex-

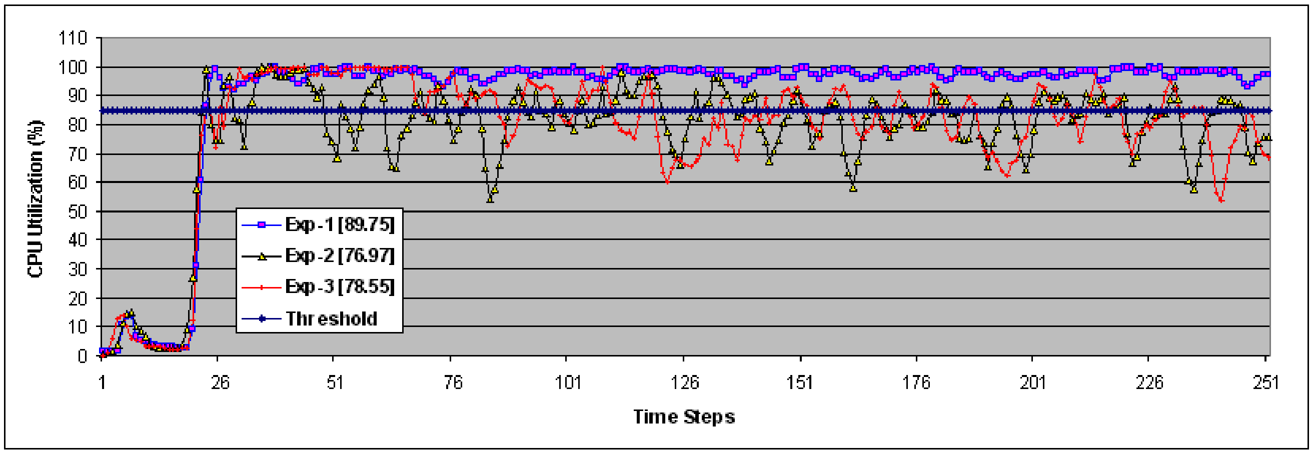


Figure 13. Server's CPU utilization measurements.

clusively on the characteristics of both the violation events and the violated policies within a single management interval; i.e., without the learning mechanisms. The third experiment (*Exp-3*) looked at the impact of action selection mechanisms based on learning from past experience in the use of policies. We used the areas occupied by the curves beyond the thresholds (85% for CPU utilization, 50% for memory utilization, and 2000 ms for response time) to compare the performance of the server relative to the base experiment (i.e., Exp-1). This provided a measure of the amount of time the system spent in "violation" states, essentially allowing us to compare performance improvement relative to the base experiment.

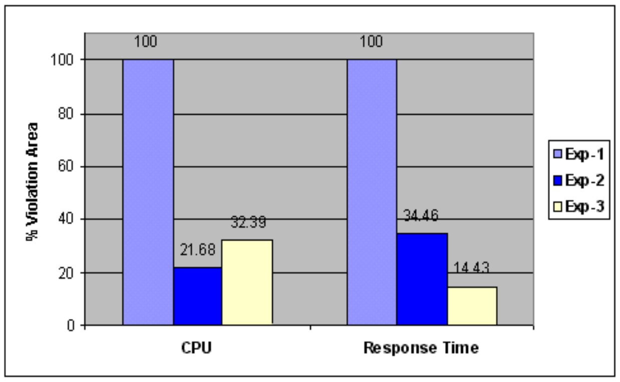


Figure 14. Area beyond the thresholds.

7.5.1 CPU Utilization

Figure 13 compares the behavior of the server in terms of CPU utilization. The number listed in square brackets beside each experiment is the average utilization for the duration of the experiment. From these results, we can see that the average CPU utilization for the base experiment (i.e., Exp-1) fell above the threshold value (i.e., 85%) whereas that of Exp-2 and Exp-3 fell below the threshold. While the main objective was to ensure that CPU utilization did not exceed 85% (which was accomplished in both Exp-2 and Exp-3), it is worth noting that action-selection mechanisms based on learning from past experience in the use of policies (i.e., Exp-3) performed slightly worse than when no learning mechanisms were enabled (i.e., Exp-2). This became more obvious when we considered the area occupied by the graphs above the thresholds relative to the base experiment as illustrated in Figure 14.

There are several reasons for this: The most obvious is probably the impact of γ-`action` (see, for example, Table 2) particularly during the initial stages of the learning process whereby the agent may be forced to spend more time *exploring* its environment (while building up the model). This may include trying actions such as γ-`action`; i.e., doing-nothing instead of performing actual adjustments to the tuning parameters to resolve QoS violations. This stage can clearly be seen from the graph of Exp-3 in Figure 13; i.e., between time-steps 26 and 70. The less obvious reason relates to the fact that the agent may have to consider multiple, and at times competing, objectives and this might be the best way of optimally meeting all the objectives. Thus, while the server may have performed slightly worse in Exp-3 than in Exp-2, the reverse was also true when considering the server's response time (see Figure 14) and throughput (see Figure 18). This is an illustration of one of the key challenges facing autonomic systems; i.e., how to negotiate between seemingly conflicting objectives: On the one hand, striving to meet customer

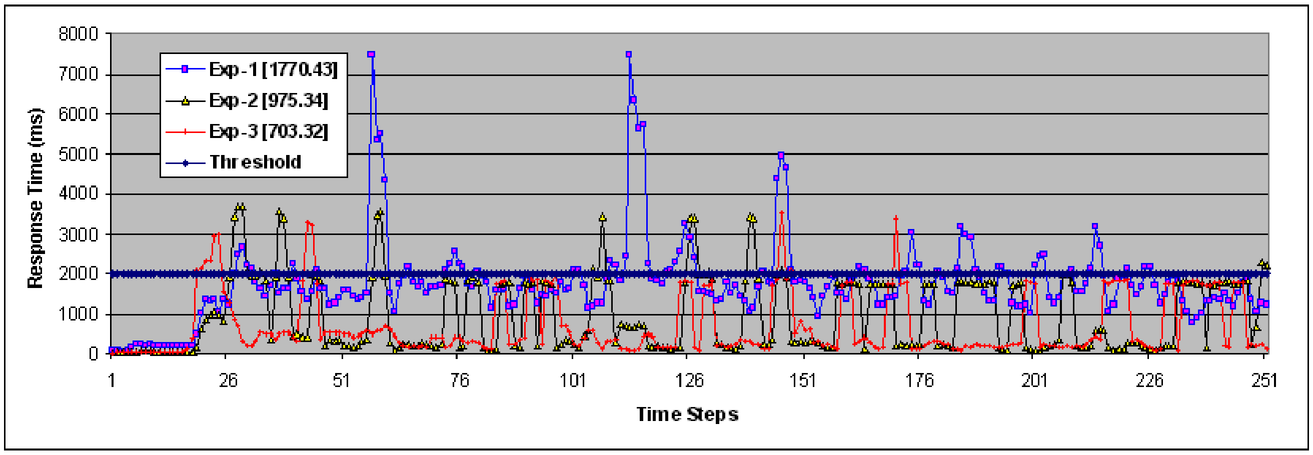


Figure 15. Server's response time measurements.

needs, in this case improving server's response time; on the other hand, trying to ensure efficient operation of systems and utilization of services.

7.5.2 Response Time

Response time measurements on the server side corresponded to the amount of time requests from non *keep-alive* connections spent on the waiting queue before they were served. The results are depicted in Figure 15 which also shows the average response for each experiment listed inside square brackets. From these measurements, one can see a significant improvement in the server's response time. This became more obvious when we computed the area above the thresholds relative to the base experiment as illustrated in Figure 14 where Exp-2 recorded at least a 65% improvement while Exp-3 recorded at least an 85% improvement in response time.

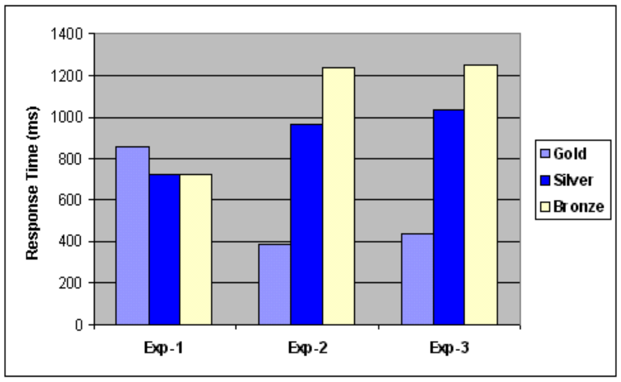


Figure 16. Client's response time.

We also compared client-side response time measurements which calculated the average time it took for a client to receive a response from the server (see Figure 16). Since no service differentiation mechanisms were enabled for the base experiment (i.e., Exp-1), the measured response was somewhat similar for gold, silver, and bronze clients. However, this changed significantly in Exp-2 and Exp-3 where gold clients response time was significantly better than that of silver and bronze clients. We also observed significant improvement in the response time of gold clients in Exp-2 and Exp-3 compared to the average of Exp-1. However, between the two experiments, there was very little difference when similar service classes were compared.

7.5.3 Throughput

Throughput measurements looked at the average number of requests serviced by the server for the duration of the experiment. The results specific to Exp-3 are shown in Figure 17: results for all the three experiments are summarized in Figure 18. Again, since no service differentiation mechanisms were enabled in Exp-1, the measurements were essentially similar for the three service classes. Furthermore, comparing the average across service classes (see the values listed inside square brackets in Figure 18), one can see that slightly more requests were serviced in Exp-1 than in Exp-2 and Exp-3. This was expected since there weren't any restrictions, for example, in terms of the server's resource utilization. In terms of the performance of individual service classes for both Exp-2 and Exp-3, the throughput measurements were consistently higher for the gold than for the silver and bronze service classes. The server also performed consistently better in Exp-3 than in Exp-2 across service classes.

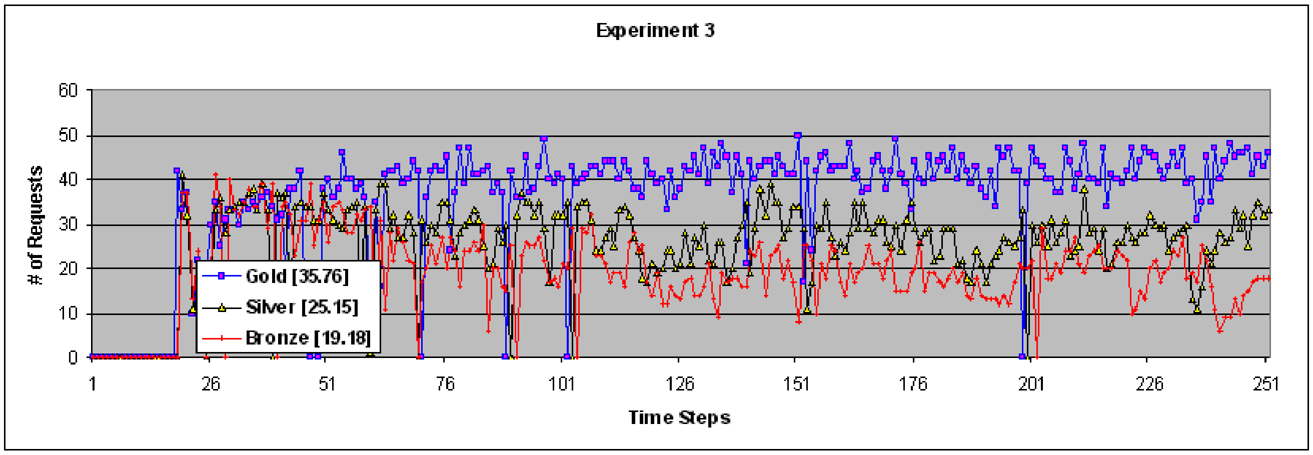


Figure 17. Server's throughput measurements.

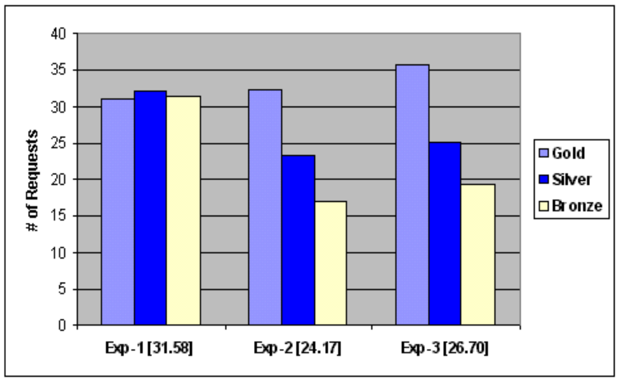


Figure 18. Server's throughput.

8 Related Work

Recently, several approaches based on Reinforcement Learning have been proposed for managing systems performance in dynamic environments. This section reviews some of the research work in this area and contrast them to our approach.

The work in [21] proposes the use of Reinforcement Learning for guiding server allocation decisions in a multi-application Data Center environment. By observing the application's state, number of servers allocated to the application, and the reward specified by the SLA, a learning agent is then used to approximate $Q_\pi(s, a)$. To address poor scalability in large state spaces, the authors initially proposed an approximation of the application's state by discretizing the mean arrival rate of page requests [20]. In their most recent work [21], they address this shortfall by proposing an off-line training, to learn function approximators using SARSA(0) [3], based on the data collected as a consequence of using a queuing-model policy, π, on-line. A key assumption is that the model-based policy is good enough to give an acceptable level of performance.

The authors in [22] propose a framework which make use of Reinforcement Learning methodologies to perform adaptive reconfiguration of a distributed system based on tuning the coefficients of fuzzy rules. The focus is on the problem of dynamic resource allocation among multiple entities sharing a common set of resources. The paper demonstrates how utility functions for making dynamic resource allocation decisions, in stochastic dynamic environments with large state spaces, could be learned. The aim is to maximize the average utility per time step of the computing facility through the reassignment of resources (i.e., CPUs, memory, bandwidth, etc.) shared among several projects.

The work in [23] proposes the use of Reinforcement Learning techniques in Middlewares to improve and adapt the QoS management policy. In particular, a Dynamic Control of Behavior based on Learning (DCBL) Middleware is used to learn a policy that best fits the execution context. This is based on the estimation of the benefit of taking an action given a particular state, where the action, in this case, is a selection of a QoS level. It is assumed that, each managed application offer several operating modes from which to select, depending on the availability of resources.

Our approach differs in several ways; First, the model of the environment is "learned" on-line and used, at each time-step, to improve the policy guiding the agent's interaction with the environment. Second, our strategy for adapting the use of policies makes use of a learning signal that is based only on the structure of the policies and should, thus, be applicable in other domains. Similarly, changing policies dynamically means that the heuristics will still work for a new set of policies. Since the state signal is dependent only on the enabled expectation policies, its structure and size can also be automatically determined once a set of policies is specified. Third, we do not make use of policies which are by themselves "models" of the system being managed. While steady-state queuing models have received significant interest in on-line performance management and resource allocation in dynamic environments, we note that most of these approaches model the behavior of the application using the mean requests arrival rate, ignoring other important characteristics. In dynamic Web environments, for example, requests to dynamic pages with database intensive queries could stress the application significantly different (in terms of server's response, resources utilization, etc.) compared to, say, requests to static pages under the same rate. Our policies, on the other hand, are simpler and do not make any assumptions about workload characteristics. Fourth, our approach does not make any assumption about the accuracy of the policies used to drive autonomic management. We view learning as an incremental process in which current decisions have delayed consequences on how the learning agent behaves in future time-steps. It is significantly important, therefore, for training to be performed on-line in order for the agent to learn from the consequences of its own decisions and, if necessary, dynamically adapt the policy guiding its interaction with the environment.

9 Conclusion

In this paper, we have proposed a strategy for determining how to best use a set of active policies to meet the different performance objectives. Our focus has particularly been on the use of Reinforcement Learning methodologies to determine how to best use a set of policies to guide autonomic management decisions. Such use of learning has significant ramifications for policy-driven autonomic systems. In particular, it means that system administrators no longer need to manually embed system's dynamics into policies that drive autonomic management. Unlike previous work on the use of action policies, for example, which required system administrators to manually specify policy priorities for resolving run-time policy conflicts, desirable behavior could be learned. It should be noted, however, that, while Reinforcement Learning offers significant potential benefits from an autonomic computing perspective, several challenges remain when these approaches are employing in real-world autonomic systems. This section looks at how we intend to address some of these challenges.

9.1 Challenges

The choice of how to model system states has significant impact on the learning process. As with many real-world systems, the state space can become prohibitively large since its size increases exponentially with the number of state metrics and their discretization. As such, storing and analyzing statistics associated with each state may require significant computation resources, which could be exceedingly costly to implement in a live system. In our current approach, we make an approximation in the representation of the system's state by mapping the conditions of the enabled expectation policies onto the state metrics. A system where each state has ten metrics, each with two possible regions (i.e., "violation" and "acceptable"), for example, would have 2^{10} (1024) possible states. We note that, in such a system, a majority of the states are likely to correspond to "acceptable" system's behavior. This is illustrated in Table 2 where out of the four states, only one state (s_1) is considered a "violation" state since it is the only state that results in the violation of the policy in Figure 3. Thus, while the size of the state space could be large, many of these states may be considered as "goal" states and, as such, would have no actions associated with them. Furthermore, it is not guaranteed that the agent would visit all the possible states during the learning process. An immediate consequence of this is a reduction in the amount of information associated with states and their transitions.

As was noted previously, the change in the system's state might be a result of external factors other than the consequences of the actions of the agent. In a Web-server environment, such transitions are often triggered by changes in workload characteristics. For example, a sudden increase in the number of clients could trigger a transition to a violation state. The fact that the state signal is not derived from such characteristics means that the learning agent can not be certain about whether or not the system's behavior at time $t+1$ is the consequence of its action at time t. We have taken the approach of excluding requests characteristics from the state signal mainly due to the stochastic nature of the interactions between these characteristics and the behavior of the system. For instance, the number of concurrent connections, the type of request (i.e., static vs dynamic), the requests rate, etc., all these could have significant ramifications on the behavior of the server. Including these characteristics as part of the state signal is likely to add significant overhead in the learning process. Excluding such characteristics, on the other hand, will not hinder the learning process since in the long run, the agent would learn about the impact of the action at s_t as the number of times the action is taken becomes large.

The decision to exclude requests characteristics from the state signal means that transitions between states could be a result of other factors. We refer to such transitions as γ-transitions (see, for example, Figure 7). We note that such transitions are more likely to originate from "acceptable" states since most of these states would have no actions associated with them. For example, a sudden increase in the number of clients requests may cause a violation in CPU utilization; i.e., a γ-transition from an "acceptable" state with no CPU utilization violation to a "violation" state. It may also be possible for such transitions to originate from "violation" states. Revisiting our example in Table 2, it might be that all three actions of the policy of Figure 3 are invalid, in which case no action could be taken while the system is in state s_1. The only possible transition, in this case, would be a γ-transition. We note, however, that such transitions are rare in comparison to those originating from "acceptable" states since it is unlikely that all state actions would fail within a single management interval. The existence of γ-transitions in the state-transition graph introduces some interesting challenges for the learning agent. First, the agent may have to decide whether doing nothing (i.e., taking a γ-action) while in state s might be better than, say, taking an action advocated by the violated policies. This may require having to learn the action-values associated with the γ-transitions (i.e., $Q_t(s, \gamma)$). Second, the learning agent may need to distinguish between two "acceptable" states if past experience shows that one state is more unstable than another. The measure of stability could be based on the characteristics of γ-transitions.

9.2 Future Work

Policy conflicts remain one of, if not, the most challenging area in policy-driven autonomic management. On the one hand, conflicts due to policy overlaps can, in most cases, be detected and corrected by analyzing static policy characteristics. On the other hand, policy conflicts which arise from dynamic characteristics specific to policy interactions can only be detected at run-time. For autonomic systems to function correctly, these kinds of conflicts need to be addressed. To what extent Reinforcement Learning could help address some of these challenges is something we hope to address in our future work.

Model-based Reinforcement Learning methods tend to be computationally demanding, even for fairly small state spaces, and could be costly when implemented in a live system. As pointed out previously, this is often due to the size of the state space as well as the computations required to process information associated with the states and actions. The key challenge then is ensuring that computational costs specific to on-line learning tasks do not hinder the learning process. In order to address this challenge, we have begun looking at how *management policies* (see Section 3.2.3) could be used to optimize resources usage during the learning process. This may include, for example, deciding on the circumstances under which computation-intensive algorithms (i.e., action-value estimations) could be executed or paused depending on the current behavior of the system. We are also interested in the use of management policies for "tuning" the behavior of algorithms to meet the resource constraints imposed by the environment. This may, for example, involve dynamically selecting the types of *updates* to be performed in order to minimize the algorithms' use of computational resources. For instance, management policies could be used to determine a reasonable value for k (which determines how many updates can be performed) in the Dyna-Q algorithm (see Algorithm 1 in Section 2).

The use of policies in autonomic computing means that the system must be able to adapt not only to how it uses the policies, but also to run-time policy modifications. In the context of where policies are used to drive autonomic management, this often means dynamically changing the parameters of the policies, enabling/disabling policies or actions within policies, or adding new policies onto an active set of policies. A key question then is whether a model "learned" from the use of one set of policies could be applied to another set of "similar" policies, or whether a new model must be learned from scratch as a result of run-time changes to the policies driving autonomic management. Our most recent work [31] has began addressing some of the questions.

References

[1] R. Murch, *Autonomic Computing*. IBM Press., 2004.

[2] R. M. Bahati, M. A. Bauer, and E. M. Vieira, "Adaptation Strartergies in Policy-Driven Autonomic Management," in *International Conference on Autonomic and Autonomous Systems (ICAS'07)*, Athens, Greece, July 2007, p. 16.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: an Introduction*. MIT Press, 1998.

[4] L. P. Kaelbing, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," in *Journal of Artificial Intelligence Research*, April 1996, pp. 237–285.

[5] R. S. Sutton, "Integrated Architecture for Learning, Planning, and Reacting based on Approximating Dynamic Programming," in *International Conference on Machine Learning*, Austin, TX, USA, 1990, pp. 216–224.

[6] A. W. Moore and C. G. Atkeson, "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time," in *Machine Learning*, vol. 13, no. 1, October 1993, pp. 103–130.

[7] J. Ping and R. J. Williams, "Efficient Learning and Planning Within the Dyna Framework," in *International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1993, pp. 281–290.

[8] R. S. Sutton, "Dyna, an Integrated Architecture for Learning, Planning, and Reacting," in *SIGART Bulletin*, vol. 2, no. 4, 1991.

[9] N. Damianou, N. Dulay, E. C. Lupu, and M. S. Sloman, "Ponder: A Language for Specifying Security and Management Policies for Distributed Systems: The Language Specification," Technical Report, Imperial College, London, UK, Version 2.1, April 2000.

[10] H. L. Lutfiyya, G. Molenkamp, M. J. Katchabaw, and M. A. Bauer, "Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-based Framework," in *International Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, Bristol, UK, January 2001, pp. 185–201.

[11] J. O. Kephart and W. E. Walsh, "An Artificial Intelligence Perspective on Autonomic Computing Policies," in *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, 2004, pp. 3–12.

[12] S. Wang, D. Xuan, R. Bettati, and W. Zhao, "Providing Absolute Differentiated Services for Real-Time Applications in Static-Priority Scheduling Networks," in *IEEE/ACM Transactions on Networking (TON'04)*, vol. 2, December 2004, pp. 326–339.

[13] T. Kelly, "Utility-directed Allocation," in *Workhop on Algorithms and Architectures for Self-Managing Systems*, San Diego, CA, USA, June 2003.

[14] P. Thomas, D. Teneketzis, and J. K. MacKie-Mason, "A Market-based Approach to Optimal Resource Allocation in Integrated-Services Connection-Oriented Networks," in *INFORMS Telecommunications Conference*, Boca Raton, FL, USA, 2000.

[15] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility Functions in Autonomic Systems," in *International Conference on Autonomic Computing (ICAC'04)*, New York, NY, USA, May 2004, pp. 70–77.

[16] M. J. Katchabaw, "Quality of Service Resource Management," Ph.D. dissertation, The University of Western Ontario, London, ON, Canada, June 2002.

[17] R. M. Bahati, M. A. Bauer, C. Ahn, O. K. Baek, and E. M. Vieira, "Policy-based Autonomic Management of an Apache Web Server," in *International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'06)*, vol. 2, no. 1, Erfurt, Germany, September 2006, pp. 21–30.

[18] R. M. Bahati, M. A. Bauer, and E. M. Vieira, "Policy-driven Autonomic Management of Multi-component Systems," in *IBM International Conference on Computer Science and Software Engineering (CASCON'07)*, Richmod Hill, ON, Canada, October 2007, pp. 137–151.

[19] R. Das, G. Tesauro, and W. E. Walsh, "Model-Based and Model-Free Approaches to Autonomic Resource Allocation," Technical Report, IBM Research," RC23802, 2005.

[20] G. Tesauro, "Online Resource Allocation Using Decompositional Reinforcement Learning," in *Association for the Advancement of Artificial Intelligence (AAAI'05)*, 2005.

[21] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in *International Conference on Autonomic Computing (ICAC'06)*, Dublin, Ireland, June 2006, pp. 65–73.

[22] D. Vengerov and N. Iakovlev, "A Reinforcement Learning Framework for Dynamic Resource Allocation: First Results," in *International Conference on Autonomic Computing (ICAC'05)*, Seattle, WA, USA, January 2005, pp. 339–340.

[23] P. Vienne and J. Sourrouille, "A Middleware for Autonomic QoS Management based on Learning," in *International Conference on Sofware Engineering and Middleware*, Lisbon, Portugal, September 2005, pp. 1–8.

[24] Apache Http Server Project. [Online]. Available: http://www.apache.org/

[25] PHP. [Online]. Available: http://www.php.net/

[26] MySQL Database. [Online]. Available: http://www.mysql.com/

[27] PHP Bulletin Board. [Online]. Available: http://www.phpbb.com/

[28] RFC2616: Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.txt.

[29] eAccelerator. [Online]. Available: http://eaccelerator.net/

[30] Linux. [Online]. Available: http://www.linux.org/

[31] R. M. Bahati and M. A. Bauer, "Adapting to Run-time Changes in Policies Driving Autonomic Management," in *International Conference on Autonomic and Autonomous Systems (ICAS'08)*, Gosier, Guadeloupe, March 2008, pp. 88–93.