Policies and Abductive Logic: An Approach to Diagnosis in Autonomic Management

Michael Tighe, Michael Bauer
Department of Computer Science
The University of Western Ontario
London, ON, N6A 5B7, CANADA
Email:{mtighe2;bauer}@csd.uwo.ca

***Abstract*—Policy-based Autonomic Management monitors a system and its applications and tweaks performance parameters in real-time based on a set of governing policies. A policy specifies a set of conditions under which one or more of a set of actions are to be performed. It is very common that multiple policies' conditions are met simultaneously, each advocating many actions. Deciding which action to perform is a non-trivial task. We propose a method of diagnosing the system to try to determine the best action or actions to perform in a given situation using Abductive Inference. We develop an original method of building a causality graph to facilitate diagnosis directly from a set of policies. We propose two alternate methods of ranking diagnosis hypotheses based on their likelihood of success. Performance of the diagnosis method is evaluated within an autonomic management system monitoring the performance of a LAMP (Linux, Apache, MySQL, PHP) server being governed by the manager. The performance of the diagnosis method is compared against previous methods used by an existing autonomic manager. The results are favourable when compared to previous methods of action selection and to the server running without the autonomic manager. A walkthrough of an example experiment using diagnosis is presented to gain additional insight into the method.**

***Keywords*-Autonomic Computing; Policy; Policy-based Management; Diagnosis; Adbuction**

I. Introduction

Autonomic Computing represents an effort to make distributed, highly interconnected and interdependent systems into self-reliant systems, capable of configuring, optimizing, healing and protecting themselves [1]. Taking a naming cue from the human autonomic nervous system, the motivation behind autonomic computing is to relieve the massive strain on human Information Technology workers from managing and configuring large systems. The task of installing, configuring, and micro-managing these systems needs to be passed on to the system itself, leaving only high level goals and objectives to be specified by human operators.

Policy-based Autonomic Management aims to fill one piece of the Autonomic Computing vision, by automating the configuration and optimization of several applications running together, in real-time. Performance metrics are monitored for running applications, and configuration parameters are tweaked in real-time to match the current environment and workload [2]. The goal is to achieve some Quality of Service (QoS) objectives [2]. The knowledge used to decide what to change and when to change it is stored within a set of *policies*. The primary type of policy in use in current autonomic management systems is the *action* policy (also called *obligation* or *expectation* policies) [3]. An action policy specifies actions for a manager to perform given that a specific set of events or conditions are present [2]. When the conditions of a policy are true and action should be taken, the policy is said to be *violated*.

In a system containing multiple policies governing the behaviour of the autonomic manager, multiple policy violations advocating many different actions are not only inevitable but are in fact commonplace. The violation of multiple policies may be the result of several discrete problems, or a single problem manifesting itself in several locations. Determining which action to perform out of the set of all actions available is a non-trivial decision [4]. Current work on selecting an action in such a situation has attempted to assign weights to actions based on a number of factors, and then execute the action with the highest weight. We propose to use abductive diagnosis [5] to try and determine the best action to perform. Abductive diagnosis uses knowledge of causal relationships between problems and causes to hypothesize about the specific cause or causes of a given set of problems [6]. This knowledge can be modeled in a bipartite graph. We introduce a method of building such a graph using the policies themselves, with no modifications or other input required. We then test this method by implementing it in an existing Autonomic Management tool.

The remainder of this paper is organized as follows: In Section II we examine related work in Policy-based Autonomic Management and Diagnosis. In Section III, we discuss an Autonomic Management tool, called BEAT, in which we have implemented our diagnosis work. Section IV describes the current method of policy action selection. In Section V, we introduce Abductive Reasoning and Diagnosis. In Section VI, we propose a method of applying Abductive Diagnosis to Policy-based Autonomic Management. In Section VII we describe the implementation of the method as well as our experiments, and present some results in Section VIII. Section IX presents an illustrative example of the diagnosis method in action. Finally, Section X provides conclusions and some thoughts on future work.

II. Related Work

Work in Policy-based Autonomic Management focuses on both the manager and the policy language itself. One such language is Ponder [7], an object-oriented, declarative policy language. It is designed as a generic language that can be used in a number of different implementations [7]. AGILE [8] is another policy language, designed for flexibility and to provide run-time adaptation of policies. It has been developed as part of a larger policy-based autonomic management system. Lymberopoulos et al [2] have developed a policy-based framework in which policy adaptation is the key focus. The authors construct a framework for network services management, with policies capable of being dynamically adapted to meet a changing workload and environment [2]. Other policy languages include PDL (Policy Description Language) from Bell Labs [9] and CIM-SPL (Common Information Model Simplified Policy Language), which is from the DMTF (Distributed Management Task Force) and is part of their larger CIM [10].

Abductive logic has been used in policy conflict detection by Bandara et al [11]. In this method, conflict detection is done prior to execution on a formalized version of the policies. Other work on diagnosis in autonomic computing has developed outside of policies, and has focused on determining the component that is the root cause of a given problem using machine learning methods. Duan and Babu [12] developed a system called *Fa* which monitors a large number of system metrics and performs diagnosis using a learned classifier. The classifier is learned via supervised learning by annotating sets of system metric values with failure states. Ghanbari and Amza [13] combine models for anomaly detection on individual components together into a single belief network, modelling the structure and causal relationships of components. Learning based on observing injected faults is performed to refine the model and the probabilities associated with the causal relationships. Also, the individual models used to detect component anomalies must be trained.

Previous work on diagnosis has not considered the use of policies. Our approach to diagnosis therefore differs from previous work in that it focuses specifically on diagnosis in policy-based autonomic management. We make use of the structure and content of the policies themselves, thus making our approach domain independent.

III. BEAT Autonomic Manager

The diagnosis algorithm has been implemented in a previously developed Autonomic Management tool, called BEAT (Best Effort Autonomic Tool). BEAT is a policy-based autonomic management framework, described in [3], [4]. Policies are used to specify how the management is performed as well as how the manager itself operates. The BEAT management system knows how to monitor and manipulate the system, and the policies provide the necessary rules to dictate how such manipulations should be carried out. These are low-level policies specifying specific actions to be taken under specific circumstances. Individual monitored system and application metrics are used to determine the situation (conditions) and actions consist of the modification of specific tuning parameters.

```
expectation policy RESPONSETIMEViolation
if (APACHE:responseTime > 2000.0) &
(APACHE:responseTimeTREND > 0.0) then
   AdjustMaxClients(+25)
     test MaxClients + 25 < 501 |
   AdjustMaxKeepAliveRequests(-30)
     test MaxKeepAliveRequests - 30 > 1 |
   AdjustMaxBandwidth(-128)
     test MaxBandwidth - 128 > 128
end if
```

Figure 1: Pseudo-code Response Time Policy

A single policy, or policy rule, consists of two main components: A set of conditions, and a set of actions. Figure 1 shows a pseudo-code version of a typical policy in BEAT. This policy describes what should occur when the response time of a web server exceeds a certain threshold. Note that this is only a representation of a policy. Actual policies in BEAT are not written in this way, and are instead built in a GUI and stored within a relational database. The policy essentially states that given that these conditions hold true, one of these actions should be performed (if CONDITIONS then ACTIONS).

Policy conditions compare some system metric to a value using a specified operator, and can be combined using standard logical operators. A single condition is not unique to one policy, but can be contained within several policies within the system. In Figure 1, the conditions are APACHE:responseTime $>$ 2000.0 and APACHE:responseTimeTREND $>$ 0.0. In these cases, the metrics being monitored are the response time of the Apache web server and the recent trend of the response time.

Policy actions specify some system or application parameter to be modified in response to the violation of the policy. For example, in Figure 1, AdjustMaxClients(+25) is an action modifying the Max Clients parameter of Apache by increasing it by 25. The action may also contain a test that must be performed and passed before execution. This could be used, for example, to prevent modifying a value beyond some hard upper and lower bounds. The AdjustMaxClients(+25) action is associated with the test MaxClients + 25 $<$ 501, thus enforcing an upper bound of 500 on the Max Clients parameter. Again, a single action may be advocated by multiple policies. In addition, a policy will specify a list of actions, with the implication that only one should be executed, but not all. The decision as to which action to execute falls on the Autonomic Manager itself.

The BEAT Autonomic Manager consists of several com-

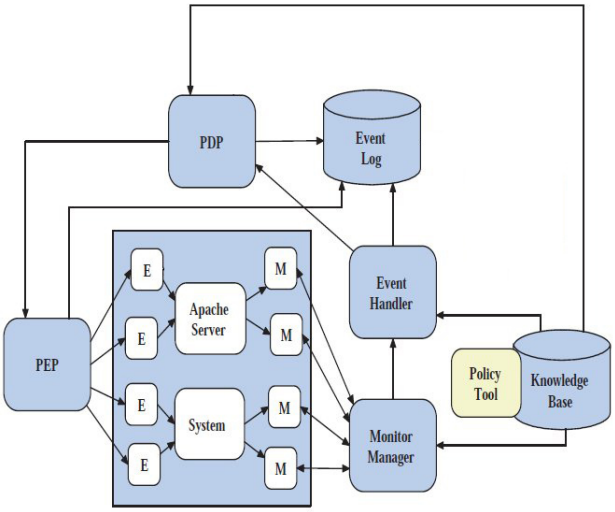

Figure 2: BEAT Autonomic Manager Architecture [3]

ponents which interact with each other to provide the full management functionality. Figure 2 [3] shows the general architecture of the system. *Monitor* components (labelled *M*) monitor the state of the system and running applications being managed, and forward this information to the *Monitor Manager*. The Monitor Manager aggregates and processes this information, and generates events which are sent to the *Event Handler*. This component then determines if the events are of interest (if they represent a violation of a policy), and forwards events to the *Policy Decision Point (PDP)*. The PDP uses the policy information to determine what, if any, action should be taken. Actions to be executed are then sent to the *Policy Enforcement Point (PEP)*, which is responsible for executing the action. The PEP determines if the action can be performed, and if so, forwards it to an appropriate *Effector* component (labelled *E*), which performs the actual modification to system and application parameters. Policies and other information are stored in the *Knowledge Base* and manipulated via a *Policy Tool*. The *Event Log* records previous events.

IV. Policy Action Selection

In a system containing multiple policies governing the behaviour of the autonomic manager, multiple policy violations advocating many different actions are not only inevitable but are in fact commonplace. The violation of multiple policies may be the result of several discrete problems, or a single problem manifesting itself in several locations. Determining which action to perform out of the set of all actions available is a non-trivial decision [4]. There are a few ways in which an action can be selected. One possibility is to simply select the first action that arises, which is essentially an arbitrary selection. This method takes nothing into account in its decision making, leaning heavily on the expertise of the policy designer, and therefore seems to be a poor choice. Another option is to weight policies and actions based on some criteria. Possible criteria include [4]:

- The *severity* of the violation, which refers to how far a threshold value on a metric has been exceeded.
- Manually assigned weights on policy conditions.
- The *advocacy* of the action, which refers to the number of violated policies advocating the same action.
- The *specificity* of the policy, which refers to the number of conditions used to trigger the policy, assuming that policies containing more conditions should be dealt with first.

These criteria and others can be used separately or in combination to provide some guidance in the action selection process. These criteria are based on intuition, and it is unclear how well they choose the best action to execute. Another possibility is to employ machine learning techniques to learn the "best" action to select in a given circumstance, based on previous experience [14], [15]. Again, this could be used in conjunction with other techniques to improve the action selection mechanism.

If an incorrect action is selected and taken, not only is time wasted before the correct action can be selected, but the modification of application tuning parameters that should not have been modified may cause further problems. This makes action selection a key problem in the performance of an autonomic manager.

V. Abduction and Diagnosis

Abductive reasoning is an alternative to deductive and inductive reasoning. This form of reasoning most closely resembles how a human diagnoses problems. Let us say that a problem consists of a set of rules, a specific case, and a result that occurs given the two. In abductive reasoning, we have the set of rules and the result, and we hypothesize about the specific case that is causing the result [6]. For example, if a doctor is diagnosing a patient, the set of symptoms experienced by the patient would be analogous to the result and the doctor's medical knowledge would be the set of rules. The doctor's diagnosis as to what the potential ailments the patient could have would be the set of specific case hypotheses. Note that unlike deduction and induction, we do not arrive at a definitive decision or conclusion; we can only build hypotheses representing what the specific case might be [6].

Peng and Reggia [6] present a formal method of representing and diagnosing an abductive reasoning problem. Given a set of *disorders* representing underlying problems, a set of *manifestations* representing observable symptoms, and knowledge of the causal relationship between the two, adbuctive methods can be used to build a diagnosis. This can be represented by a graph, which we will call a *Causal Network*, containing both the disorder and manifestation sets. An edge from a disorder to a manifestation indicates that the disorder may cause the manifestation, although

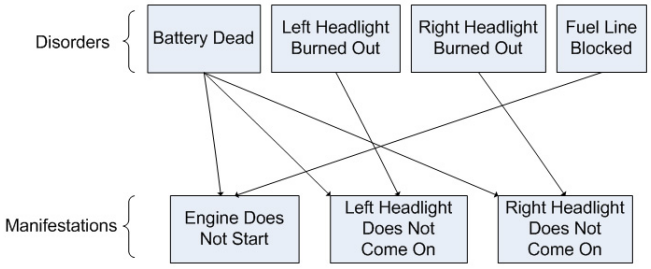

Figure 3: Causal Network Example (based on example from Peng and Reggia [6])

it is important to note that it may not. A disorder can cause multiple manifestations, and a manifestation may be caused by many different disorders. Given a set of currently present manifestations and the causal network, diagnosis can be performed to build a set of hypothesis disorder sets that could explain the manifestations. It is impossible to guarantee that a definitive diagnosis can be obtained. The best that can be achieved is the construction of a set of hypotheses. Each hypothesis contains a set of disorders that fully explain the present manifestations, but determining which hypothesis is correct or even which hypotheses are more likely to be correct is a non-trivial task.

A simple example is given in Peng and Reggia [6] describing a causal network for the diagnosis of automotive problems. It uses a small set of disorders and manifestations and presents the causal associations between them that form the causal network graph. The disorders include *battery dead*, *left headlight burned out*, *right headlight burned out*, and *fuel line blocked*. The manifestations are *engine does not start*, *left headlight does not come on*, and *right headlight does not come on*. Figure 3 shows the causal network for these disorders and manifestations, including the causal associations between them.

A *cover* of a given set of present manifestations is a set of disorders such that each present manifestation can be caused by at least one disorder in the set. Each cover represents a single hypothesis solution, giving one potential explanation for the manifestations. Peng and Reggia [16] suggest that simpler covers are more likely to be true than complex ones. It is then these simple covers that we wish to find when diagnosing a problem. There are several different suggested criteria for judging the simplicity of a cover. A *single-disorder* cover is a cover that consists of only a single disorder. An *minimal* cover contains the minimal number of disorders required to cover all present manifestations. An *irredundant* cover is a cover where each disorder causes at least one manifestation that no other disorder in the cover causes. A *relevant* cover is a cover that contains no disorders that are not a cause of at least one present manifestation. These criteria create increasingly broad sets of covers as we move from single-disorder to relevant covers. The set of single-disorder covers for a set of manifestations is a subset of the set of minimal covers, which is a subset of the set of irredundant covers, which is finally a subset of the set of relevant covers.

Of these criteria, irredundancy seems intuitively to be the best choice. Single-disorder covers are unnecessarily restrictive, and clearly insufficient in situations where more than one problem (disorder) can occur simultaneously. Minimal covers are also too restrictive, as it is easy to imagine a case where a minimal cover is clearly not the most likely explanation of the manifestations. For example, in medical diagnosis, a minimal cover may consist of a single rare disease, where another cover may exist containing two common diseases. Clearly the minimal cover is less likely in this case. Relevant covers, on the other hand, represent the other extreme in which far too many covers are accepted as plausible. Irredundant covers will therefore be used for diagnosis.

An algorithm has been developed by Peng and Reggia in [6] to diagnose the problem by constructing the set of all irredundant covers of the present manifestations. The actual diagnosis algorithm, presented in Figure 4 is quite simple. The algorithm starts with a set of hypotheses, and is given the set of current manifestations. It then iterates through each manifestation (in no particular order), revising the hypothesis set each time to represent all irredundant covers of the new manifestation as well as previously added manifestations. Once all manifestations have been accounted for, the algorithm is complete.

```
1: hypothesisSet = {∅}
2: while moreManifestations do
3:    m_new = nextManifestation;
4:    hypothesisSet = revise(hypothesisSet, causes(m_new))
5: end while
6: return hypothesisSet
```

Figure 4: Diagnosis Algorithm

The heart of the algorithm is the *revise* method. The method, which accepts the current set of hypotheses as well as the set of causes for the manifestation being added, results in a new hypothesis set that contains all irredundant covers of the set of manifestations that have been processed, including the new one. It does this by first finding all hypotheses in hypothesisSet that are also irredundant covers of the new manifestation, and leaving them unchanged. It then modifies any remaining covers so that they also cover the new manifestation by adding new disorders to them. Finally, any duplicate or irredundant covers created in the last step are removed. Details of the algorithm and revise method can be found in [6].

Let us return to our basic example of automotive diagnosis from [6], illustrated in Figure 3, and take a high level look at the diagnosis algorithm in action. Let us say that we

are currently observing all of the possible manifestations, that is, *engine does not start*, *left headlight does not come on*, and *right headlight does not come on*. We start with an empty hypothesis set in line 1 of the algorithm, and lines 2 to 5 proceed to loop through each presently observed manifestation, one at a time. The first manifestation, *engine does not start*, is retrieved in line 3. Line 4 revises our current hypothesis set, which is empty, by including the causes of *engine does not start*, which are *battery dead* and *fuel line blocked*. This results in a set of two hypotheses, namely, either *battery is dead* or *fuel line is blocked*.

Next we add the *left headlight does not come on* manifestation, and revise the hypothesis set with its causes (*battery dead* and *left headlight burned out*). The revise method first picks out *battery is dead* as a cover of both the original manifestation and the new one, and decides to leave it in the hypothesis set. Next, it modifies the second cover, *fuel line is blocked*, so that it covers the new manifestation. This is done by adding the disorder *left headlight burned out*. Other covers are possible, but would not be irredundant. This results in a new set of hypotheses, where either *battery is dead* is true (which would cover both manifestations itself), or *fuel line is blocked* and *left headlight burned out* are both true.

Finally, we add the last manifestation, *right headlight does not come on*, using its causes (*battery dead* and *right headlight burned out*). This brings us to our final set of hypotheses, which explains (covers) all three presently observed manifestations. We still have only two hypotheses. Either the disorder *battery dead* is true, which itself explains all three manifestations, or *fuel line blocked*, *left headlight burned out* and *right headlight burned out* are all true simultaneously. Both of these hypotheses are irredunant covers of the set of presently observed manifestations. Logically, it makes sense that given the manifestations, either the battery is dead or the fuel line is blocked and both headlights are burned out.

VI. Mapping Policies to a Causal Network

In order to perform diagnosis, a Causal Network containing potential disorders, manifestations, and their relationships must be constructed. We do this based on existing information contained within the set of policies governing the Autonomic Manager, at run-time. In this way, diagnosis can be used for action selection without requiring any extra diagnosis-specific information to be added. This method of bridging the gap between policies and abductive diagnosis constitutes our main original contribution.

Each policy contains a set of conditions and a set of actions. These conditions and actions are not unique, and the same conditions and actions will be used by many different policies. The manifestations can be derived from the conditions, in that each condition is directly mapped to a single manifestation. If the condition is true, then the equivalent manifestation is considered present. Disorders are derived from the policy actions, with each action being used to build a single disorder. Since an action is intended to correct some parameter that is thought to be set incorrectly for the current environment and workload, then that parameter being incorrectly set can be considered the underlying disorder causing the manifestations. For example, if an action specifies that the Max Clients parameter of the Apache server should be increased by 25, then the disorder derived from such an action would be *Apache Max Clients too low*.

Associations between the generated manifestations and disorders can be easily derived from the policies as well. If a policy containing the condition used to derive a certain manifestation also advocates the action used to derive a certain disorder, then that manifestation could potentially be caused by the disorder and should be associated with it. Since conditions and actions are replicated across many different policies, this method results in a fairly well connected Causal Network. Diagnosis can then be performed using this Causal Network, essentially finding the action or actions that can potentially cause all present conditions to no longer be true, thus eliminating all policy violations. Figure 5 gives a psuedo-code version of the algorithm.

```
1:  disorderSet = {∅}
2:  manifestationSet = {∅}
3:  causalRelationships = {∅}
4:  for all policies p do
5:     for all conditions c in p do
6:        m = Manifestation(c)
7:        if m not in manifestationsSet then
8:           manifestationSet += m
9:        end if
10:       for all actions a in p do
11:          d = Disorder(a)
12:          if d not in disorderSet then
13:             disorderSet += d
14:          end if
15:          causalRelationships += (d, m)
16:       end for
17:    end for
18: end for
19: return  hypothesisSet
```

Figure 5: Policy Mapping Algorithm

The size of the Causal Network depends on the number of policies deployed in the system, and the level of redundancy in the policy conditions and actions. To look at one extreme, if each policy contains completely unique conditions and actions, it will result in a graph with many nodes and very few connections. If, on the other hand, the conditions and actions are replicated throughout many different policies (which is the usual case), the graph will have fewer nodes

and be well connected. Even with a large number of policies, the Causal Network will remain a simple bipartite graph, and should scale well as the number of policies increases.

```
expectation policy RESPONSETIMEViolation
if (APACHE:responseTime > 2000.0) &
(APACHE:responseTimeTREND > 0.0) then
   AdjustMaxClients(+25)
     test MaxClients + 25 < 501 |
   AdjustMaxKeepAliveRequests(-30)
     test MaxKeepAliveRequests - 30 > 1 |
   AdjustMaxBandwidth(-128)
     test MaxBandwidth - 128 > 128
end if

expectation policy CPUViolation
if (CPU:utilization > 85.0) &
(CPU:utilizationTREND > 0.0) then
   AdjustMaxKeepAliveRequests(-30)
     test MaxKeepAliveRequests - 30 > 1 |
   AdjustMaxBandwidth(-128)
     test MaxBandwidth - 128 > 128
end if
```

Figure 6: Pseudo-code CPU Policy

Figure 7 shows an example Causal Network derived from example policies in Figure 6. The CPUViolation specifies actions to be taken to lower CPU utilization in the event that it exceeds a threshold value of 85%. For simplicity in the diagram, the Manifestation nodes that would be generated for the trend conditions (APACHE:responseTimeTREND and CPU:utilizationTREND in Figure 6) are omitted, as they would be identical to the nodes derived from APACHE:responseTime and CPU:utilization, respectively. In the actual causal network, they would be present.

Since each action must pass an associated test before it can be executed, diagnosis must generate a list of hypotheses, with the first one that passes its associated tests being executed. The set of all hypotheses obtained from diagnosis must therefore be ordered from most likely to be effective to least likely to be effective. The only ranking method currently implemented is to sort the hypotheses based on the number of disorders they contain. This can either be done in ascending or descending order, causing the system to favour either hypotheses containing fewer disorders or more disorders, respectively. This translates to the system either preferring to execute fewer actions when using ascending or preferring to execute many actions with descending.

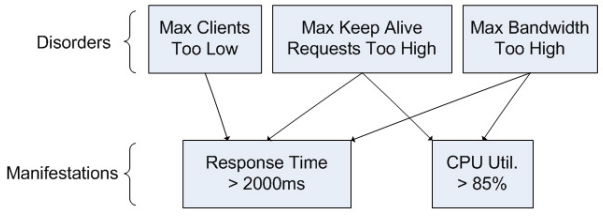

Figure 7: Policy derived Causal Network

Let us see how this causal network could be used for some very basic diagnosis. Say that we observe the manifestation *Response Time* $>$ *2000ms*. We can see that any of the disorders could be the cause, that is, our set of hypotheses includes three single disorder hypotheses (*Max Clients Too Low*, *Max Keep Alive Requests Too High*, and *Max Bandwidth Too High*). Now, let us say that we also observe the manisfestation *CPU Utilization* $>$ *85%*, and update our set of hypotheses using the causes of this manifestation. This reduces our set of hypotheses to two, namely, *Max Keep Alive Requests Too High* and *Max Bandwidth Too High*. You may note that a hypothesis containing both *Max Bandwidth Too High* and *Max Clients Too Low*, or both *Max Keep Alive Requests Too High* and *Max Clients Too Low* would also be a potential cover of the present manifestations. In both cases, however, the disorder *Max Clients Too Low* would be redundant, and we only wish to look for *irredundant* covers, as discussed in Section V.

Hypotheses containing disorders must then be translated back into something useful to the autonomic manager, that is, a list of actions or sets of actions to perform. For each hypothesis, each disorder contained within it is used to look up the original action used to build the disorder. The actions for a single hypothesis are grouped together, and if executed, the entire group must be executed together, since all disorders contained in the hypothesis are required to cover the present manifestations. Using our example, the two single disorder hypotheses of *Max Keep Alive Requests Too High* and *Max Bandwidth Too High* would be translated back into the actions AdjustMaxKeepAliveRequests(-30) and AdjustMaxBandwidth(-128), as seen in Figure 6.

VII. Experiments

The diagnosis action selection method was implemented in the BEAT Autonomic Manager discussed in Section III. Modifications to BEAT were made in the Policy Decision Point (PDP) component, inserting diagnosis in place of the existing method. We compared the diagnosis method to other methods of selecting policy actions by configuring BEAT to manage a web server, and measuring its performance under a stressful workload. Performance is measured with the autonomic manager using the action selection method previously used in BEAT (describe in Section IV), with two variations of the newly developed diagnosis algorithm, and with the server running without intervention by the manager. Policies are specified with the goal of maintaining specific response time, CPU utilization, and memory utilization ranges, and the methods of action selection can be compared on how well they achieve these objectives. Service differentiation will be used and controlled by the autonomic manager. Incoming requests to the server are divided into three service classes, namely, gold, silver and bronze, with gold being given highest priority and bronze lowest.

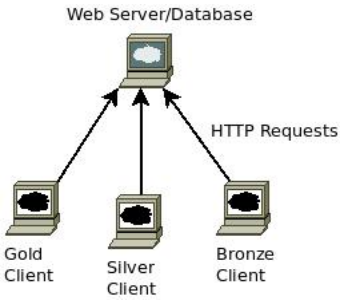

Figure 8: Experimental Setup

A. Test Environment

Figure 8 shows the basic setup of our experiments.

Server: The server machine hosts a web server running a PHP bulletin board application [17]. The application makes use of a database, also running on the server machine. The server is a LAMP stack (Linux, Apache, MySQL, and PHP), with the BEAT Autonomic Manager installed.

Clients: There are three client machines responsible for generating the workload for the server. Each machine represents one service class, namely, gold, silver and bronze. Requests sent by the gold machine are to be given highest priority, and requests sent by the bronze machine are to be given the lowest. In a real world implementation, requests could be divided into classes based on a pricing plan, importance as a part of a larger system, or some other prioritization scheme. Load was generated using Apache JMeter 2.3.4 Load Generator [18].

B. Systems Under Test

Four configurations will be contrasted with each other to evaluate the performance of the diagnosis algorithm. These include the system without the aide of BEAT, with BEAT enabled and using the previously developed action selection method, and finally with two different versions of the diagnosis algorithm.

Policies Disabled: A base configuration with the BEAT autonomic manager disabled, and with differentiated services disabled (all requests are treated equally). This will provide a frame of reference for judging the performance improvement offered by the autonomic manager with each form of action selection.

Weighted Actions: Section IV outlines a set of criteria that can be used to guide action selection. These criteria, (severity, specificity, weight and advocacy), are implemented in BEAT [15] and combined together to determine a total weight for each action, with higher weighted actions being given priority. Details of this can be found in Bahati et al. [15]. For the purposes of this experiment, we will refer to this as the *Weighted Actions* method.

Diagnosis with Fewer Disorder Priority (Diagnosis - Fewer): The first of the two forms of the diagnosis algorithm is Diagnosis with Fewer Disorder Priority. The algorithm itself for both forms is identical. The difference is in the ordering of hypotheses. In this form, hypotheses containing fewer disorders are ranked higher than hypotheses with more disorders. Essentially, this makes the assumption that the simplest hypothesis is most likely the correct one. In many cases this will result in a single action being taken, but it is not necessarily always the case.

Diagnosis with Many Disorder Priority (Diagnosis - Many): This second form of the diagnosis algorithm reverses the ordering of the first. It makes the assumption that taking multiple actions will be more likely to be successful than taking a single action. As such, hypotheses containing more disorders will be given priority over those containing fewer. The diagnosis algorithm is otherwise unchanged from Diagnosis with Fewer Disorder Priority. This will often result in multiple actions being taken.

C. Measures of Performance

Four metrics will be measured to determine the relative performance of each version of the Autonomic Manager.

- *Apache Response Time (Server)* - This is the response time of the Apache web server as measured from the server itself. This value is extremely important, as it is the measure by which the autonomic manager itself determines how well the server is performing. It measures response time by continuously requesting a single page from the web server and measuring the time it takes to receive it. It does not use the KeepAlive option, meaning that a new connection must be opened for each request. It is also independent of the service differentiation mechanism used for requests received from external machines.
- *CPU Utilization* - This is the percentage of the CPU currently in use on the server machine. This does not refer to the amount of CPU being used by the web server only, but rather the total CPU usage.
- *Memory Utilization* - This is the percentage of the total memory that is in use on the server. Like the CPU Utilization metric, this represents total memory usage for the entire server machine.
- *Client-side Response Time* - This is the response time as measured by the client machines. The time taken to complete each request (from the time the request is sent until the entire page has been received) is recorded. These requests make use of the KeepAlive option, meaning that requests sent by a single 'user' attempt to re-use the same existing connection. The set of all client-side request response times can be divided into the three service classes (gold, silver and bronze) to analyze the effects of service differentiation.

D. Workload

All three client machines ran identical workloads, and were started and stopped simultaneously. The workload was designed to overload the system to a point where without the aid of the autonomic manager, the CPU is running at 100% and server-measured response times are over the 2 second threshold. The workload started with a single thread (or user), and ramped up linearly to a total of 25 threads (or users) over a period of 8 minutes. Each thread continuously performed a small loop consisting of a "think-time" delay of 750-1250ms, and a request to a page randomly chosen from 24 dynamic (PHP generated) pages offered by the PHP Bulletin Board application running on the server. A request included retrieving the HTML page as well as all other resources (images, etc.) contained on the page. This continued for one hour, at which point the test was halted. Each thread used the KeepAlive option, thus attempting to reuse its existing connection to the server as much as possible to avoid reconnecting.

E. Policy Goals

The policies are designed in such a way as to maintain certain performance objectives, or goals. These typically consist of a threshold value on a measured metric within the system. The duty of the autonomic manager is to achieve these goals as best it can. These goals roughly translate to the conditions of the policies. When the conditions are violated, the policy actions are intended to attempt to push the metric back under the threshold. Without going into detail as to the specific policies and policy actions, the following are the general goals of the set of policies used in this experimentation:

- Apache HTTP server response time, as measured from the server should be below 2 seconds.
- The CPU Utilization should be below 90%. If utilization falls below 85%, then more CPU resources should be used, if needed. Essentially, the system should make use of as much CPU as it can up to the 90% threshold.
- Memory Utilization should be below 50%.
- Priority should be given to the gold, and then the silver and finally the bronze service classes in that order.

VIII. Results

The experiment was performed identically with each of the four systems under test (Policies Disabled, Weighted Actions, Diagnosis - Fewer, and Diagnosis - Many). Each test was run for exactly one hour, and repeated a total of 5 times. Averages and standard deviations were calculated for each run and averaged over the 5 repeats of the experiment.

Table I shows the metrics measured on both the server and client machines. These include the response time of the Apache web server (as measured by the mechanism described in Section VII-C), CPU Utilization and Memory Utilization. The values shown are the average values for an entire run. The metric averages are then averaged across all 5 replications of the experiment. Figure 9 shows the Apache Response Time data for all experiment replications as a box plot, and figure 10 is a box plot the CPU Utilization for all experiments replications.

	Disabled	Weighted	Diag. Fewer	Diag. Many
Apache Resp.	3336ms	1195ms	1031ms	1163ms
CPU Util.	98.3%	74.4%	82.5%	82.4%
Memory Util.	22.3%	24.6%	24.0%	24.1%
Gold Avg.	2182ms	1798ms	1389ms	1465ms
Silver Avg.	2228ms	4021ms	3920ms	3827ms
Bronze Avg.	2192ms	4742ms	5543ms	5239ms

Table I: Average Results

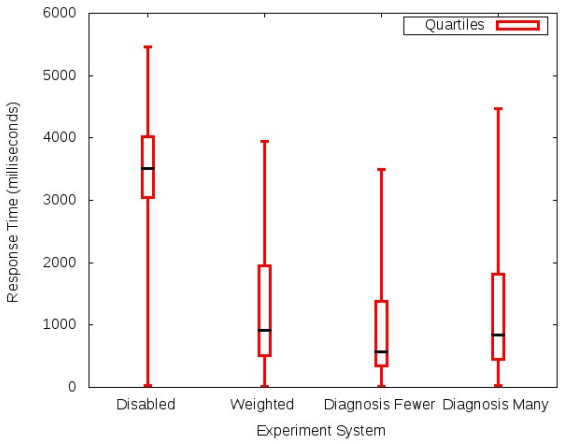

Figure 9: Apache Response Time Box Plot

Judging by the measured response times of the Apache web server, we can easily see that the three tests performed with the autonomic manager outperform the system with the manager disabled. CPU utilization also comes down under the threshold value, while memory utilization increases by a trivial amount and stays well below threshold levels. The response times for the three action selection methods are

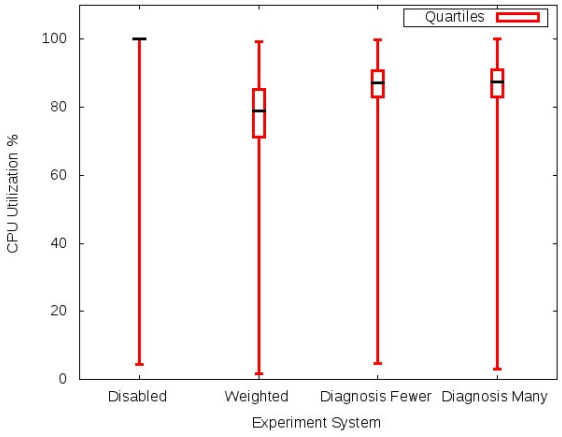

Figure 10: CPU Utilization Box Plot

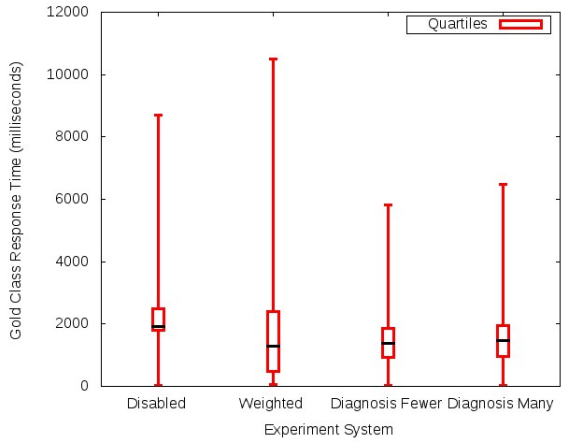

Figure 11: Gold Response Time Box Plot

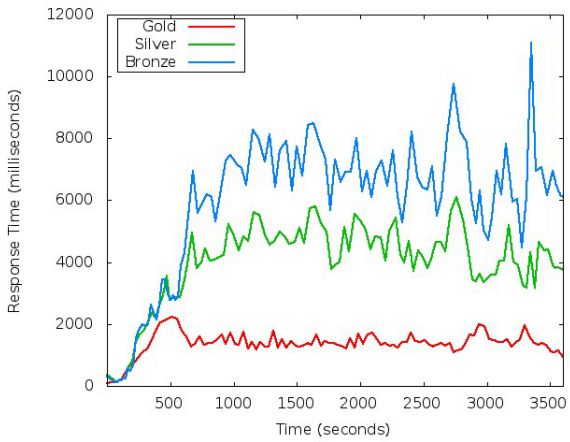

Figure 12: Client Response Times for Diagnosis Fewer

similar, with Diagnosis Fewer (favouring hypotheses with fewer disorders, or fewer actions to take) beating out the other two, which match up fairly evenly. Both variations of the diagnosis algorithm also make better use of the CPU, as outlined in the goals of our set of policies in Section VII-E, without going over the 90% threshold.

The Gold, Silver, and Bronze response times in table I are the client-side response times of the gold, silver, and bronze client machines, respectively. Response times experienced by the client machines show a different side to the performance of the web server than those measured by the server-side response time monitor. As mentioned in Section VII-C, the server-side response time metric does not use KeepAlive, while the client machines do. This means that the server monitor needs to open a new connection for each request, and as such potentially wait in a queue again. Another difference comes from the effect of service differentiation on the client requests. The most important of the client response time measures is that of the gold service class, as the silver and bronze classes should be sacrificed to maintain its performance. The test is designed to put the server under stress, and as such we should see response times of the silver and bronze classes sacrificed to maintain the performance of the gold class. Figure 11 is a box plot of the Gold Class Response Time data for all experiment replications. Both diagnosis algorithms perform better than weighted action selection on gold response time, as well as silver response time, with diagnosis favouring fewer actions having the edge. Figure 12 shows the gold, silver, and bronze response times for the system using the diagnosis algorithm favouring fewer actions, for a single repetition of the experiment. Service differentiation is clearly visible in this graph, as the system keeps the gold class consistent at the expense of silver and bronze.

Figure 13 compares the Apache response times for weighted action selection and diagnosis favouring fewer

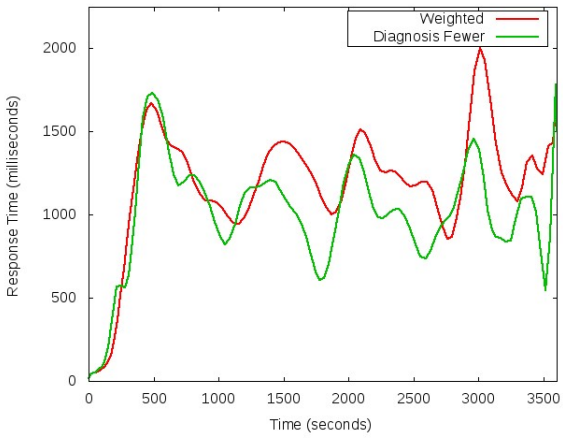

Figure 13: Apache Response Time

actions, for a single repetition of the experiment. The graphed curves are Bezier curve approximations of the actual data, in order to more clearly show the difference between the performance of the two methods. A Bezier curve is a parametric curve approximation of the data used to smooth the data. The data shown is from a single experiment, not averaged over all 5 repetitions, and represents results consistent with all experiments.

Table II shows the same metrics as table I, except only for the overload period of the experiment. That is, the ramp up time to the maximum load of 25 clients per machine (for a total of 75 clients) is excluded, leaving only the time

	Disabled	Weighted	Diag. Fewer	Diag. Many
Apache Resp.	3722ms	1300ms	1095ms	1247ms
CPU Util.	99.9%	78.0%	86.8%	87.2%
Gold Avg.	2333ms	1872ms	1398ms	1463ms
Silver Avg.	2365ms	4442ms	4387ms	4257ms
Bronze Avg.	2336ms	5404ms	6657ms	6233ms

Table II: Average Results - Max Load Only

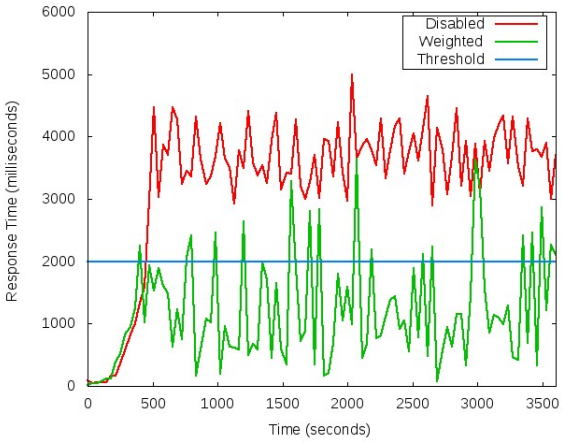

Figure 14: Apache Response Time Over Threshold

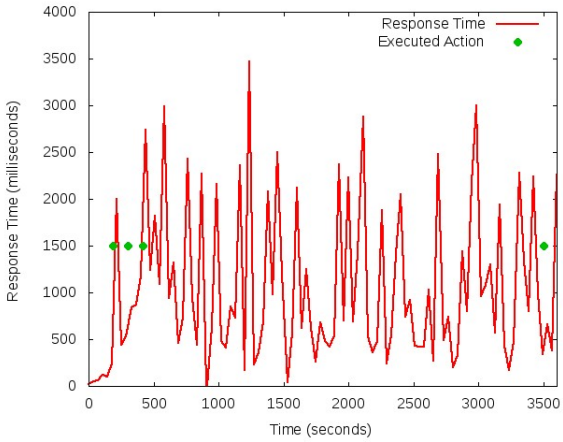

Figure 15: Example Run with Diagnosis - Response Time

	Disabled	Weighted	Diag. Fewer	Diag. Many
Apache Resp.	290.30	14.94	11.19	16.19
CPU Util.	3.65	0.04	0.13	0.15

Table III: Server Metrics Area Over Threshold

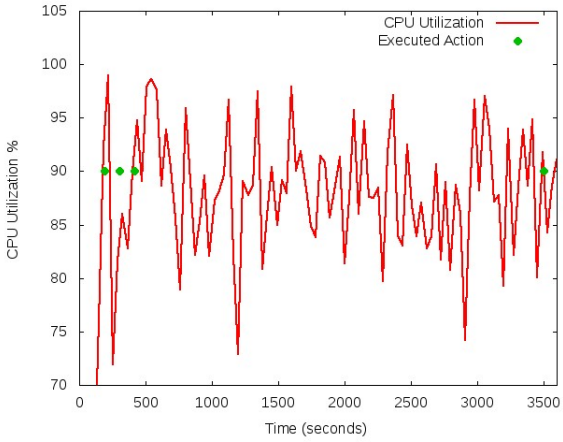

Figure 16: Example Run with Diagnosis - CPU Utilization

period when the server was operating under maximum load (75 clients total). The results are slightly different in value to those of the entire run, but values in comparison with each other remain consistent.

Another way to look at the data is to examine not the averages but the amount of time the value is over the specified threshold, and by how much. This can be done by calculating the area of the curve over the threshold. Figure 14 shows the measured response time of the Apache web server with the manager disabled and with weighted action selection, compared to the threshold value of 2000ms, for a single repition of the experiment. The area between the threshold value and the response time curve above it provides a useful measure of how well the goals of the policies are being achieved. Table III contains these values. The values shown are averaged over the 5 experiment repetitions. The system with the manager disabled exceeds the thresholds of both Apache response time and CPU utilization far more than with the manager enabled. Diagnosis favouring fewer actions comes out on top yet again, with weighted actions and diagnosis features multiple actions coming in second and third, respectively. Note that since the units of response time and CPU utilization are not the same, we cannot compare directly between the response time and CPU utilization area over threshold values.

IX. Example Run with Diagnosis

To help illustrate how the autonomic manager behaves, particularly when using the diagnosis algorithm for action selection, we will take a look at an example experiment run and go into some detail at a few points of interest. The information examined is from a single experiment repetition, but is typical of all of the experiments. We will look at the diagnosis algorithm favouring hypotheses with fewer disorders (fewer actions to take). It is difficult to tell the direct consequences of each decision made and action executed, since a very large number of actions are executed throughout the experiment and the workload is dynamic. Nevertheless, some thoughts as to why the diagnosis algorithm performs slightly better than weighted action selection can be derived from such an analysis.

Figures 15 and 16 show the response time and CPU utilization metrics for an example run of the system using diagnosis favouring fewer actions. Four points of interest are marked on each graph and explained in some detail, in order from left to right (sequential order in time).

Point 1: The first violations occur at around the three minute mark (180 seconds).

1) Apache Response Time Violation
2) Apache CPU without Response Time Violation
3) PHP Response Time Violation
4) MySQL Response Time Violation

To begin with, this is an interesting combination of violation events. The first, third and fourth violations are all triggered by the same conditions, namely, the response time of the web server exceeding 2000ms and having an increasing trend. The difference lies in the set of advocated actions by each violation. Each policy advocates actions related to a different component of the system, namely, the Apache web server, the PHP cache, and the MySQL database. What makes this particular set of violations interesting is the second violation, namely, the Apache CPU without Response Time Violation. The conditions for this violation are CPU utilization above 90% and rising, and the response time of the web server being below 200ms, a contradiction with the conditions of the other policy violations. Clearly the response time cannot be both above 2000ms and below 200ms. What probably has occurred is that both states were present at some point in the interval between the last time the policies were checked for violations and this time. This interval during these experiments was 10 seconds.

The diagnosis algorithm then attempts to build hypotheses that can explain the situation we are seeing, even though we know that these particular violations do not represent a single snapshot of the state of the system, but rather what has occurred over the last 10 seconds. This is not necessarily a bad thing, as such seemingly contradictory information may in fact lead the diagnosis algorithm to finding a better solution by eliminating some extraneous actions or even including actions that may not have been considered otherwise. Whereas the weighted action selection will select an action based on applying some importance to each policy and each action independent of each other, the diagnosis algorithm takes into account the entire situation in its decision making. This may account for some of why the diagnosis algorithm performs better than weighted action selection.

The diagnosis algorithm builds the set of all possible actions or sets of actions that can cover the given policy violations. In this case, the first three are single actions that cover every condition in each policy. Since in this example the algorithm is favouring hypotheses with fewer disorders (fewer actions to take), these single actions are ranked first.

1) Decrease the maximum number of clients in Apache
2) Decrease the maximum number of KeepAlive requests in Apache
3) Decrease the maximum bandwidth, which compromises the performance of lesser service classes to maintain the performance of higher classes, with gold being the highest and bronze the lowest.

Hypotheses indicating that more than one action should be performed are ranked lower. An example of such a hypothesis is one that advocates both decreasing the MySQL Key Buffer size and increasing the cache memory available for PHP at the same time. The ordering of hypotheses containing the same number of actions is arbitrary, and is based on the order in which the violations are given to the algorithm and how the algorithm operates. It can be considered essentially random. Nevertheless, actions are attempted in the order they are sent to the PEP. In this particular case, the first action (decrease the maximum number of clients) was not performed because its associated test failed (the parameter was already at its lowest possible value). The second action, decreasing the maximum number of KeepAlive requests, was performed.

Point 2: After the first set of violations, a large number of the policy violation situations consist simply of the three Response Time Violations.

1) Apache Response Time Violation
2) PHP Response Time Violation
3) MySQL Response Time Violation

Since all three of these violations share the same conditions, the resulting diagnosis is simply a list of all actions advocated by the three policies, because any of these actions will cover all of the conditions of all three. Since the diagnosis algorithm performs no ordering of the actions within itself, it will build the same set of potential actions as the weighted action selection method, except it will make no attempt to determine which is more likely. As such, it will probably make a similar, if not slightly worse decision. The tests attached to the actions also make a difference in which action is selected, as all tests for an action must pass before the action can be executed. This means that several higher ranked actions may be skipped before reaching an action that can be performed, potentially neutralizing some of the effect of ordering.

Point 3: Another common policy violation situation occurs at the 417 second mark. At this point, we see a combination of both response time related violations and CPU utilization violations.

1) Apache Response Time Violation
2) PHP Response Time Violation
3) MySQL Response Time Violation
4) Apache CPU Utilization Violation
5) PHP CPU Utilization Violation
6) MySQL CPU Utilization Violation
7) Apache CPU and Response Time Violation

We have already seen the response time violations. All three contain the same conditions but advocate actions related to different components of the system. The three CPU Utilization violations (4, 5 and 6) are similarly related. All three have CPU utilization above 90% and an upward CPU utilization trend as their conditions, but they each advocate different actions. The Apache CPU and Response Time policy violation is triggered by a combination of both web server response time conditions and CPU utilization conditions, and advocates actions to be taken in the case that both the response time is above 2 seconds and CPU

utilization is above 90%. This policy attempts to dictate what should occur when more than one type of violation exists, and the set of actions advocated by it is actually a subset of the actions already advocated by the other policies. Such a policy is essentially trying to simulate some sort of diagnosis, and is likely rendered obsolete by the diagnosis algorithm. Nevertheless, it is in use at the moment and taken into consideration in diagnosis. The following is the list of actions or sets of actions returned by the diagnosis algorithm.

1) Decrease the maximum number of KeepAlive requests in Apache
2) Increase the cache size used for PHP pages
3) Decrease the maximum bandwidth
4) Increase the MySQL thread cache size and increase the number of Apache clients
5) Decrease the maximum number of clients in Apache and increase the MySQL key buffer size
6) Increase the MySQL thread cache size and key buffer size
7) Decrease the maximum number of clients in Apache and increase the MySQL query cache size
8) Increase the MySQL query cache size and thread cache size

As before, hypotheses containing fewer actions to perform are preferred. Only one of these will be executed, and they will be attempted in the order listed. Again, the ordering within hypotheses containing the same number of actions is essentially random.

Point 4: At around the 59 minute mark another interesting policy violation situation occurs.

1) Apache CPU Utilization Violation
2) PHP CPU Utilization Violation
3) MySQL CPU Utilization Violation
4) Apache without both CPU and Response Time Violation

We have already seen the three CPU Utilization policy violations. The fourth, Apache without both CPU and Response Time, indicates that response time is within normal constraints (below 2 seconds), and that CPU utilization is also below the violation threshold of 90%. Clearly, as we saw earlier with response times, this contradicts the other three policy violations, again most likely due to the 10 second window in which violations can occur before they are processed. The question then becomes, how should this be interpreted? This is by no means a trivial question. Should the violations indicating that the CPU utilization is over 90% be trusted or the one indicating that it is below be trusted? In weighted action selection, one of these two options will be chosen. With diagnosis, however, both options will be combined to find some solution that satisfies both, thus taking into account all of the information received. Such a difference in approach may be at least partially responsible for the improved performance of the diagnosis algorithm. In this case, a set of four hypotheses is generated, each containing two actions to perform.

1) Decrease the maximum number of clients in Apache and increase the maximum bandwidth
2) Decrease the maximum number of KeepAlive requests in Apache and increase the maximum bandwidth
3) Increase the cache size used for PHP pages and increase the maximum bandwidth
4) Increase the MySQL thread cache size and increase the maximum bandwidth

As before, the actions were attempted by the PEP in the order shown, and in this case, the very first set of actions passed its tests and was performed.

X. Conclusions and Future Work

A diagnosis approach using adbuction has been proposed to help the autonomic manager decide which action to take in the case of multiple policy violations. The approach uses the policies themselves to build a Causal Network, which is then used to perform diagnosis. The diagnosis algorithm was implemented in the BEAT Autonomic Manager [3] for testing.

We examined the performance of a web server without the aid of the autonomic manager, with the manager using weighted action selection, and using diagnosis. From the results presented here, we can conclude that the diagnosis algorithm performs at least as well as the previous method of action selection (weighted action selection). CPU utilization for all three action selection methods stays below the threshold, but the two diagnosis methods make use of more CPU resources than weighted action selection, keeping closer to the threshold. Diagnosis favouring multiple actions performs similarly to weighted action selection, except on the actual measured client response times, where it has an edge on gold and silver service class response times. Diagnosis favouring fewer actions beats out the other methods across the board, although not by a significant margin. This indicates that the use of the diagnosis algorithm to select an action in the case of multiple policy violations makes better decisions than the previously developed weighted action selection methods, more closely achieving the overall goals of the policies, that is, keeping metrics such as CPU and Response Time within specified thresholds. Although the improvement offered by diagnosis was minor, in a larger scale experiment it may become more pronounced. Further experimentation is required to fully evaluate the method.

The advantage that diagnosis favouring fewer actions has over diagnosis favouring multiple actions seems to indicate that simpler explanations of the given set of policy violations (hypotheses containing fewer disorders) are more likely to be correct, an example of Occam's Razor [19]. The decision making advantage enjoyed by diagnosis over weighted action selection may be due to the fact that diagnosis essentially attempts to use all available information together

to make a decision, while weighted action selection pits each option against each other. This is a subtle yet potentially interesting distinction.

The diagnosis method is not a strict alternative to weighted action selection, and future work could investigate the combination of these methods. The criteria for policy and action weighting could be used to build probabilities into the causal network. The policies themselves should also be examined, as a simpler set of policies may be possible when using diagnosis. Finally, in order to fully evaluate and drive development of these techniques forward, some larger scale implementation and testing is likely necessary.

REFERENCES

[1] J. Kephart, D. Chess, I. Center, and N. Hawthorne, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] L. Lymberopoulos, E. Lupu, and M. Sloman, "An Adaptive Policy-based Framework for Network Services Management," *Journal of Network and Systems Management*, vol. 11, no. 3, pp. 277–303, 2003.

[3] R. Bahati, M. Bauer, E. Vieira, O. Baek, and C. Ahn, "Using Policies to Drive Autonomic Management," in *WoWMoM 2006. International Symposium on a World of Wireless, Mobile and Multimedia Networks.*, 2006, pp. 475–479.

[4] R. Bahati, M. Bauer, and E. Vieira, "Policy-driven Autonomic Management of Multi-component Systems," in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. ACM New York, NY, USA, 2007, pp. 137–151.

[5] M. Tighe and M. Bauer, "Mapping Policies to a Causal Network for Diagnosis," *ICAS 2010, International Conference on Autonomic and Autonomous Systems*, pp. 13–19, 2010.

[6] Y. Peng and J. Reggia, *Abductive Inference Models for Diagnostic Problem-solving*. Springer, 1990.

[7] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "Ponder: A Language for Specifying Security and Management Policies for Distributed Systems: The Language Specification," *Imperial College Research Report DoC*, 2000.

[8] R. Anthony, "Policy-centric Integration and Dynamic Composition of Autonomic Computing Techniques," in *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, 2007.

[9] J. Lobo, R. Bhatia, and S. Naqvi, "A policy description language," in *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999, pp. 291–298.

[10] Distributed Management Task Force, "Common Information Model Simplified Policy Language," http://www.dmtf.org/standards/cim_spl/, June 2010.

[11] A. Bandara, E. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, 2003, pp. 26 – 39.

[12] S. Duan and S. Babu, "Guided problem diagnosis through active learning," in *Intl. Conf. on Autonomic Computing*, 2008, pp. 45–54.

[13] S. Ghanbari and C. Amza, "Semantic-driven model composition for accurate anomaly diagnosis," in *Autonomic Computing, 2008. ICAC'08. International Conference on*, 2008, pp. 35–44.

[14] J. Bigus, D. Schlosnagle, J. Pilgrim, W. III, and Y. Diao, "ABLE: A Toolkit for Building Multiagent Autonomic Systems," *IBM Systems Journal*, vol. 41, no. 3, pp. 350–371, 2002.

[15] R. Bahati, M. Bauer, and E. Vieira, "Adaptation Strategies in Policy-Driven Autonomic Management," in *ICAS 2007, International Conference on Autonomic and Autonomous Systems*. IEEE Computer Society Press, Washington, DC, USA, 2007, p. 16.

[16] Y. Peng and J. Reggia, "Plausibility of Diagnostic Hypotheses: The Nature of Simplicity," in *Proceedings of AAAI-86*, 1986, pp. 140–145.

[17] "PHP Bulletin Board," http://www.phpbb.com/, January 2011.

[18] "Apache JMeter," http://jakarta.apache.org/jmeter/, January 2011.

[19] "Merriam-Webster Online Dictionary, Occam's Razor entry," http://www.merriam-webster.com/dictionary/Occam's razor, May 2009.