2009

Diagnosis in Policy-Based Autonomic Management

Michael Tighe

Diagnosis in Policy-Based Autonomic Management

(Spine Title: Diagnosis in Policy-Based Autonomic Management)

(Thesis format: Monograph)

by

Michael Tighe

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

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. Performance of the diagnosis method is measured by implementing diagnosis into an existing autonomic management application and monitoring the performance of a LAMP (Linux, Apache, MySQL, PHP) server being governed by the manager. The results are favourable when compared to previous methods of action selection and to the server running without the autonomic manager.

Keywords: autonomic computing, autonomic management, policy, abduction, diagnosis

Acknowledgements

I would like to acknowledge first the help of my supervisor, Dr. Michael Bauer, for pointing me in the right direction and providing guidance as I worked my way through my thesis. I would like to thank Raphael Bahati for his work on the BEAT Autonomic Manager which was vital to my work, as well as his always available assistance in my efforts to build my experiments. Finally, I'd like to give a nod to my fellow students and DiGS members for being an excellent sounding board for both ideas and the venting of frustrations.

Contents

List of Tables

List of Figures

Chapter 1 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 [20]. Taking a naming cue from the human autonomic nervous system, the motivation behind autonomic computing is to relieve the massive strain on human IT workers from managing and configuring large systems. As systems continue to increase in size, the feasibility of effectively managing them manually continues to drop [18]. The task of installing, configuring, and micromanaging 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 [21]. The knowledge used to decide what to change and when to change it is stored within a set of *policies*. A Policy instructs the autonomic manager on what to do in different circumstances [20].

The BEAT (Best Effort Autonomic Tool) Autonomic Manager uses policies to manage several applications on a server machine, such as an Apache HTTP server and MySQL database [14]. The policies specify a set of conditions with regards to measured performance metrics, and a set of potential actions to take when the conditions are met. When the conditions are met, the policy is said to be *violated*, and require corrective action to eliminate the violation. For example, a *Response Time Violation* policy could specify that when the measured response time of a web server is above two seconds, then either the maximum number of clients should be increased, or the number of connections to keep alive should be decreased, or the maximum bandwidth should be adjusted. It is the responsibility of the autonomic manager to decide which of the possible actions to perform.

Determining which action is best to perform given a set of potential actions is not trivial. This issue is compounded when multiple policies are violated simultaneously, each advocating the execution of several actions. This is the problem we attempt to tackle in this thesis. 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 to try and determine the best action to perform at the moment, based on the set of policies and the present policy violations. 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 [23]. This knowledge can be contained in a bipartite graph. We introduce a method of building such a graph using the policies themselves, with no modifications or other input required. The policies alone are used as input to automatically infer causal relationships and store these within a graph structure. Diagnosis can then be performed using the current policy violations to help determine which action or actions should be performed. The performance of the diagnosis method of action selection can be measured by implementing the method in the BEAT autonomic manager, and testing it against previous methods. How well the autonomic manager achieves the objectives of it's policies should be an indicator of how well the diagnosis method of action selection performs.

1.1 Main Contributions

This thesis presents a new method of selecting actions using abductive diagnosis. Specifically, a method of translating a set of policies into a causal network graph for use in a diagnosis algorithm developed by Peng and Reggia [23]. This method constructs a complete graph from pre-existing policy information by inferring causal relationships from condition and action pairings within the policies. It requires absolutely no modification to the existing policy definition or specific set of policies. After diagnosis has been performed, the resulting hypotheses can be directly translated back into corrective actions to be performed.

1.2 Outline

The remainder of this thesis is organized as follows: Chapter 2 covers **Background**, including an overview of policies, policy-based management, and the BEAT autonomic manager. Chapter 3, **Abduction and Diagnosis**, dives into diagnosis using abduction and an algorithm for achieving it. Chapter 4, **Implementation**, describes how the abductive diagnosis algorithm was implemented in the BEAT autonomic manager. This includes our proposed method of building a graph of casual relationships from a set of policies. Chapter 5, **Experiments**, details the testing and evaluation of the BEAT autonomic manager using abductive diagnosis. Finally, Chapter 6, **Conclusions and Future Work**, provides some final conclusions and presents ideas and thoughts for future research in the area.

Chapter 2
Background

2.1 Autonomic Computing

Autonomic Computing is quickly becoming an area of extreme interest in the realm of Computer Science. Its broad reaching goal is to create distributed computer systems that are completely self-reliant. The term Autonomic Computing intentionally borrows from the human autonomous nervous system, which controls many aspects of the body automatically and subconsciously, including heart rate, breathing and digestion. As the size and complexity of systems increases and these systems become more and more intermingled with other systems throughout the world, they are becoming ever more difficult to manage and maintain [18]. At some point these systems will grow beyond what can be effectively handled by even the most skilled IT professionals, if that point has not already been reached [18, 20]. The only real solution is to build systems that operate on their own, with minimal human intervention. Such systems must be capable of installing and configuring themselves, dynamically optimizing their own performance, and operating in a constantly changing environment with other such systems [20]. Human IT staff will specify high level goals and objectives, and let the system figure out to achieve them. This is the vision of Autonomic Computing, which IBM refers to as a *grand challenge* [18].

Autonomic Computing can be broken down into several separate, non-trivial behaviours. These include self-configuration, self-optimization, self-healing, and selfprotection, and together they comprise the ultimate goal of self-management [20]. Self-configuration is the first step in the lifecycle of an autonomic system. It involves the initial installation and configuration of a system, but does not stop there. The system must also be able to install and configure new components dynamically, reconfigure itself to meet new goals, or integrate with other existing components and systems[20]. Manual installation and configuration of large systems is a cumbersome, tedious, and error-prone process, thus making self-configuration an essential goal of Autonomic Computing. Self-optimization handles the next logical step in self-management. An Autonomic System will consist of many applications, each with its own set of parameters to be tweaked and tuned for optimal performance in any given situation. The system must therefore be able to adjust these parameters automatically and dynamically in an attempt to achieve optimal performance though changing workloads and environments [20]. Large systems are bound to encounter failures of various types and severity, and an Autonomic System must be able to handle these failures. This ability falls under the heading of self-healing [20]. Problems must be detected, diagnosed, and appropriate actions need to be taken to repair or reconfigure the system to overcome them [20]. Finally, an Autonomic System must be self-protecting, in that it must be able to defend itself against malicious attacks as well as cascading failures [20].

Autonomic Computing research can be broken down in another way, as it is in Kephart et al. [19], outlining three high level research areas. These areas are autonomic elements, autonomic systems, and human-computer interactions [19]. Autonomic elements are the primary components of an Autonomic System. They are the individual self-managing parts of the overall system, each responsible for managing one or more related computing resources and each exhibiting the previously outlined behaviors of self-management [19]. These are the pieces of the puzzle, with the complete puzzle being the autonomic system. The autonomic system is the collection of all autonomic elements, each interacting in some structured way to meet human defined high-level goals and objectives [19]. The manner in which these goals and objectives are expressed to the system by humans falls under the heading of human-computer interactions [19]. All three of these areas present unique challenges to achieving the ultimate goal of Autonomic Computing. Autonomic elements must be fully capable of configuring, optimizing, healing, and protecting themselves, while at the same time must be able to interact with other autonomic elements within the larger autonomic system to achieve the goals of the individual element as well as the system in general. There are countless problems to be tackled before Autonomic Computing is realized. That being said, even small advancements towards the ultimate vision can help ease the increasing difficulty of systems management [19].

2.2 Policies

The concept of a policy is vital to Autonomic Computing as a whole. There are a wide range of definitions of a policy, many of which will be at work simultaneously in a full autonomic system. In the most basic sense, a policy is how a human informs the autonomic system how to behave [19]. They are written by humans in some defined language and interpreted by the autonomic system. The variation in definitions then comes from what types of things are specified in the policy. Are high-level goals and objectives expressed or low-level instructions? The answer of course is that both are required at different places within an autonomic system, as well as other levels in between the two extremes. At the highest level of an autonomic system, business goals will be expressed as written policies which the system will attempt to achieve. At lower levels, policies specify actions to be taken given a certain situation. This is where the majority of the work with policies has been focused thus far [19]. The main advantage provided by the use of policies is that they allow for the decoupling of specific behaviours and strategies from the underlying management software [17]. Instead of hard coded behaviours requiring modifications and recompiling to enact a change, policies are read and interpreted in real-time and can therefore change dynamically while the system is running. This provides the flexibility needed by the autonomic management system to to quickly and easily adapt to changing situations [17].

In order for a human to specify a policy, there must exist a language to write the policy in. One such language is Ponder [17], an object-oriented, declarative policy language. Ponder allows the specification of policies for both management and security [17]. It is designed as a generic language that can be used in a number of different implementations [17]. AGILE [10] 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.

The primary type of policy in use in current autonomic management systems is the *action* policy (also called *obligation* or *expectation* policies) [14]. An action policy specifies actions for a manager to perform given a specific set of events or conditions to be present [21]. Action policies are condition-action rules designed to ensure that specific conditions or requirements are met by taking action when they are not, evaluating as an "if condition then action" statement [14]. When the conditions of a policy are true and action should be taken, the policy is said to be *violated.* Other types of policies, such as policies specifying the behaviour of the management system itself, or the configuration of the management system, can also be specified.

2.3 Policy-Based Autonomic Management

Policy-Based Autonomic Management is exactly as it sounds. An autonomic manager manages a system using a set of policies as instructions on how to behave under given conditions [11]. Most work of this nature involves the use of low level action, or obligation, policies as input for the management of one or more applications running on a server machine. The goal is generally to achieve some performance or other quality of service (QoS) objectives through the constant monitoring and tweaking of application metrics and parameters, respectively [21]. For example, the response time of an Apache web server and the CPU load of the system could be monitored and compared to some threshold value. If the values exceed the given thresholds, an action could be taken to optimize the performance of the Apache web server by reducing the value of the *KeepAlive* parameter in Apache, which controls the number of requests allowed to maintain persistent connections with the server at a single time.

Policies are an excellent choice for user input to the autonomic manager as they separate the specific rules from the underlying code responsible for executing them. Policies can be swapped or even modified at run time without the need for modifications to the code or even the restarting of the autonomic manager [10]. In [21], policy adaptation takes a central role. The authors construct a framework for network services management, making use of policies written in the Ponder policy language. These policies can be dynamically adapted to meet a changing workload and environment [21]. Policy adaptation can be achieved either through the modification of parameters in a policy (such as threshold values), dynamically enabling or disabling policies, or by using machine learning to select or modify policies based on previous experience [21]. The policy abstraction also makes policies easier to write and maintain than, for example, embedding the logic directly into the source code[10].

Policy-based Autonomic Management systems generally follow a basic high level architecture. *Monitor* components (also called *sensors*) exist that are responsible for gathering metrics from running applications or from the machine, such as response time or CPU load [15]. This data is possibly transformed or expanded on to provide some additional information, such as averages over a certain time period or trend analysis [10]. A policy component uses the metrics provided by the monitor(s), typically received as events, to trigger policies, which in turn contain actions to perform on either the running applications being managed or the autonomic manager itself. Such changes are executed by one or more *effector* components [15, 11].

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 [13]. 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, and therefore seems to be a poor choice. Another option is to weight policies and actions based on some criteria. Possible criteria include [13]:

- 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 [15, 12]. 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.

2.4 Related Work

2.4.1 BEAT Architecture

The BEAT (Best Effort Autonomic Tool) Autonomic Management system is a policy-based autonomic management framework, described in [11, 12, 13, 14]. Policies are used to specify how the management is performed as well as how the manager itself operates. These policies are manipulated via a graphical user interface and stored in persistent storage known as the *Knowledge Base*. The management system consists of several interacting components, each with a specific task. The system and applications being managed are monitored, decisions are made by the autonomic manager using knowledge stored in the Knowledge Base as policies, and changes to system and application tuning parameters are made as deemed necessary. The ultimate goal of the system is to achieve some non-functional performance or other quality of service requirements.

Policies

The BEAT Autonomic Management system uses user-specified policies to store knowledge about how to manage the system. The management system knows how to monitor and manipulate the system, and the policies provide the necessary rules to dictate the how such manipulation 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 and actions consist of the modification of specific tuning parameters. They are specified via a Policy Tool, in this case a graphical user interface providing the necessary functionality to create and manipulate policies. Changes can be made in real time while the system is running, without the need to restart the management system, thus exploiting one of the main benefits of the use of policies. Policies in the BEAT system come in three varieties:

Configuration Policies can be used to specify initial configuration settings of applications or to start or install applications or services.

Expectation Policies specify condition-action instructions with the goal of maintaining some performance or other Quality of Service (QoS) objectives. Events are triggered when the conditions of an expectation policy are violated, and actions specified within the policy are then executed to attempt to return the policy to a non-violated state.

Management Policies deal with the administration of the autonomic manager itself. Actions can be specified to modify parameters, enable or disable policies, modify policies in some other way, or for diagnosis or other required analysis.

```
expectation policy RESPONSETIMEViolation(PDP,PEP)
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 2.1: Pseudo-code Policy

At the highest organizational level, policies are contained within policy groups. These groups are used to divide policies into logical groups, such as the set of policies pertaining to each application under management. A single policy may belong to more than one group. A single policy, or policy rule, consists of two main components: A set of conditions, and a set of actions. Figure 2.1 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). The policy also has other properties, such as a name, a target, a subject, and a flag indicating whether or not the policy is enabled. The target specifies what the policy applies to, such as a specific host in a network, and the subject refers to the management component responsible for handling the policy violation.

As stated above, a policy contains a set of one or more Conditions. A condition is an expression based on some monitored system metric which evaluates to a boolean value. In the case of the BEAT system, each condition compares a metric to some value using a specified operator. For example, a condition could look for the ResponseTime metric of the Apache web server to be greater than two seconds, as does the first condition in Figure 2.1. Multiple conditions can be present within a policy, joined by standard logical operators. In our example, a second condition that the trend of the response time value must be greater than 0 is connected to the first condition with a logical AND. When the conditions section evaluates to true, then the policy is said to be violated, and some action should be performed. A single condition may be and likely will be reused in several different policies.

Policies contain zero or more Actions. An action specifies some system or application parameter to be modified in response to the violation of the policy. In a more general sense, the action specifies a function to be executed, along with any required parameters. Again, a single action may be advocated by multiple policies. Boolean tests can be associated with an action to ensure that the action makes sense in the current environment. For example, let us say that the parameter controlling the maximum number of clients that the Apache web server can serve at a time should not be increased to any higher than 500. The third action in Figure 2.1 attempts to increase the maximum number of clients by 25. Given our ceiling of 500, a test should first be performed to ensure that the new value will be less than 501. Multiple tests can be associated with an action, all of which must evaluate to true for the action to be allowed to execute. Which action or actions are executed is the decision of the autonomic manager.

Components

The BEAT Autonomic Management system consists of several components which interact with each other to provide the full management functionality. Figure 2.2 [14] shows the general architecture of the system. *Monitor* components 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 violation 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, which performs the actual modification to system and application parameters. The following is a more detailed explanation of each component of the management system.

Knowledge Base: The Knowledge Base contains all policy and rule information for the management system, as well as other data describing, or collected from, the system. Such data could include statistics or trend analysis, machine learning data, etc. Policy information stored within the Knowledge Base is generally loaded and cached by other components, with notifications being sent out to inform them of any run-time changes.

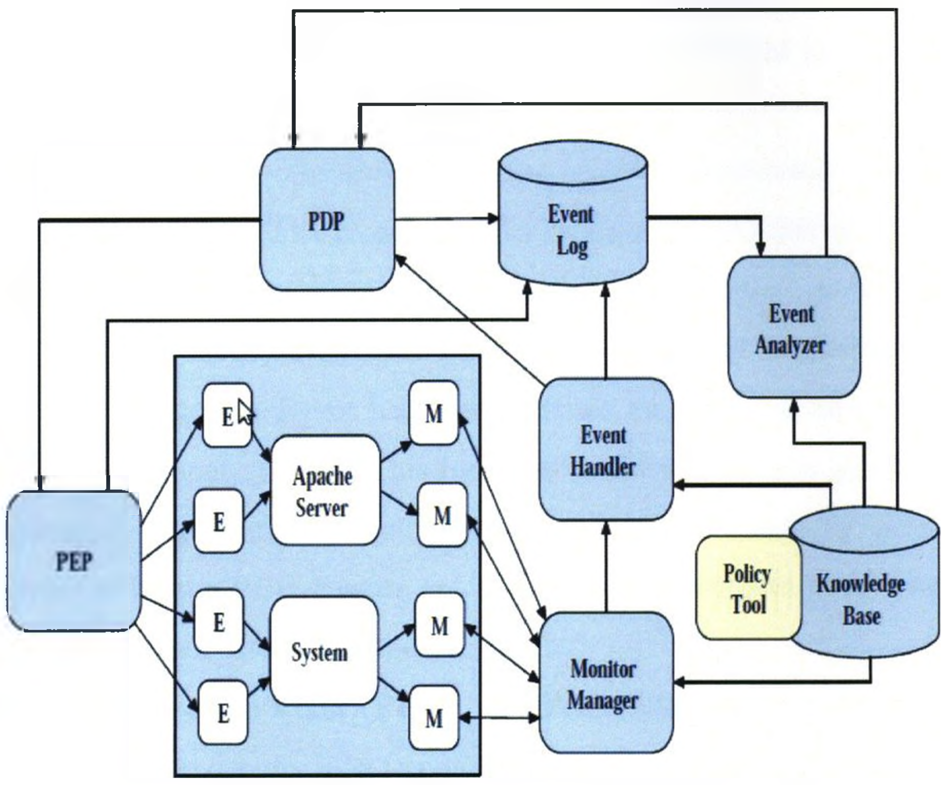

Figure 2.2: BEAT Autonomic Manager Architecture [14]

Monitor: A Monitor component collects metric information from the system or running applications for use by the management system. It generates events to be consumed by other components. There may be many monitors in use in an autonomic management system, each responsible for monitoring a specific set of metrics.

Monitor Manager: The Monitor Manager handles the configuration and processing of data from individual monitors. Monitor configuration parameters include scheduled times for monitoring to occur and monitoring frequency, among others. The Monitor Manager gathers metric information from the individual monitors and forwards it to the Event Handler.

Event Handler: The Event Handler handles monitoring events received from the Monitor Manager. The Event Handler determines whether or not the events received represent a violation of an expectation policy condition, and forwards the event to the PDP if it does. Events are also sent to the Event Log for persistent storage.

Event Log: The Event Log is responsible for recording all events that occur within the management system. This includes monitoring events generated by the Monitor Manager, expectation policy violations, decisions made by the PDP, and actions executed by the PEP. Events are written to a short term in-memory log as well as to a log file in persistent storage.

Policy Decision Point (PDP): The Policy Decision Point receives events from the Event Log and decides how to enforce any present policy violations. It must determine which policies have in fact been violated, and given the actions advocated by the set of violated policies, decide which action should be taken. This is the component in which the diagnosis algorithm will be implemented, since this is where all decisions are made regarding the execution of actions based on policy violations.

Policy Enforcement Point (PEP): The Policy Enforcement Point is responsible for executing policy actions. Once the PDP has made a decision on what should be done, an ordered list of policy actions are sent to the PEP for enforcement. The PEP must perform any tests associated with an action to ensure that the action is valid in the current environment, and subsequently call upon the appropriate Effector for executing the action.

Effector: Effector components perform the actual changes to the system. Many Effectors may exist, responsible for making modifications to specific parameters or sets of parameters. For example, one Effector could be responsible for modifying all parameters in the Apache web server. The PEP evokes the effector needed to execute the required action.

Event Analyzer: The Event Analyzer is responsible for analyzing the history of the system, including any policy violations and decisions made by the PDP, for further use. The goal is to be able to exploit previous experience to make better decisions on policy enforcement, or to predict future policy violations. The PDP could be modified to make use of data from the Event Analyzer to assist in its decision making process.

Chapter 3
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. It refers to the process of building hypotheses to explain given observations [9]. 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 [23]. 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 [23].

3.1 Diagnosis using Abductive Inference

According to the Merriam-Webster dictionary, diagnosis is the "investigation or analysis of the cause or nature of a condition, situation, or problem" [3]. In diagnosis, we are given a problem and need to determine what the cause of the problem may be. This can be achieved using abductive inference, or reasoning. 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 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. Since a manifestation could be caused by many different disorders, and a disorder can cause many manifestations but is not guaranteed to always cause the same ones or even the same number, 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 [23] 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 3.1 shows the causal network for these disorders and manifestations, including the causal associations between them.

3.1.1 Parsimonious Covering Theory

Parsimonious Covering Theory is presented as a formal method of performing diagnosis using abductive inference, and providing some measure of determining a set of hypotheses that is more plausible than others [22]. A formal model is defined to en-

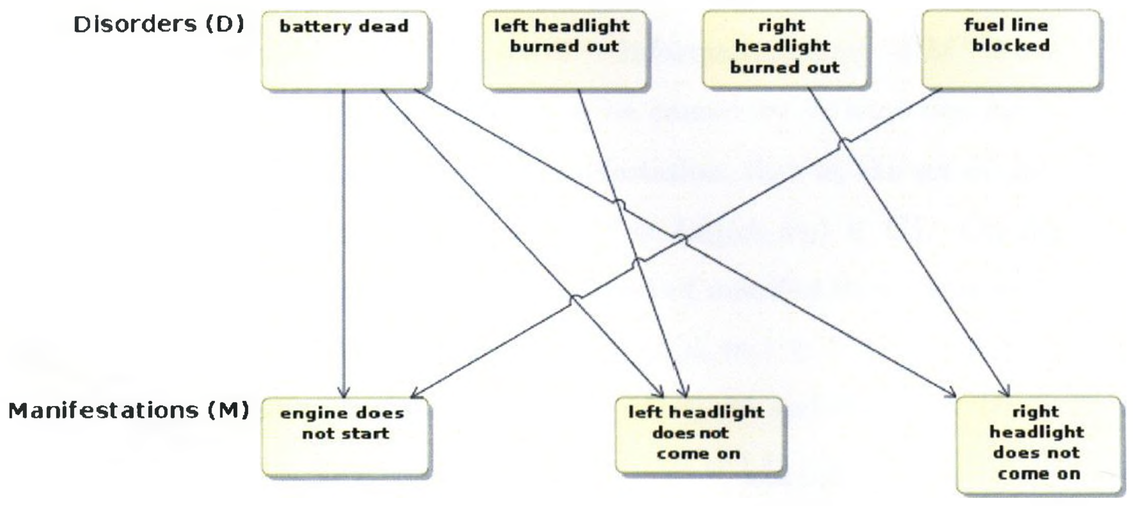
Disorders (D)
battery dead
left headlight burned out
right headlight burned out
fuel line blocked
Manifestations (M)
engine does not start
left headlight does not come on
right headlight does not come on

Figure 3.1: Causal Network Example (based on example from Peng and Reggia [23])

capsulate the causal knowledge about the disorders and manifestations. This model explains only the simplest form of causal network, with only disorders, manifestations, and edges between the two. More complicated models can include other features such as intermediate states or probabilities.

A diagnostic problem P is defined in Peng and Reggia [22] as $\langle D, M, C, M^+ \rangle$, where

$D = \{d_1, \ldots, d_n\}$ is a finite non-empty set of disorders;

$M = \{m_1, \ldots, m_n\}$ is a finite non-empty set of manifestations;

$C \subseteq D \times M$ is a relation with $domain(C) = D$ and $range(C) = M$;

$M^+ \subseteq M$ is a subset of M;

The causal relationship between sets D and M is contained in C, with $\langle d_i, m_j \rangle \in C$ iff d_i may potentially cause m_j. The set M^+ represents the set of manifestations currently present. This will be different in each specific diagnosis case, while D, M, and C will remain constant. To specifically relate this model back to our original definition of abduction, M represents all possible observations, C represents the set of rules or knowledge, D represents all possible specific cases, and M^+ represents the specific set of observations in the current problem.

We can now define a *cover* of a set M^+. Informally, a cover of M^+ is a set $D_I \subseteq D$ such that each manifestation in M^+ can be caused by at least one disorder in D_I. We can define the set of causes of a manifestation, that is, the set of disorders that may cause a manifestation, as $\mathit{causes}(m_j) = \{d_i | \langle d_i, m_j \rangle \in C\}$. On the flip side, the set of effects of a disorder, that is, the set of manifestations that may be caused by a disorder, is defined as $\mathit{effects}(d_i) = \{m_j | \langle d_i, m_j \rangle \in C\}$. These definitions can be extended to the sets of manifestations $M_J \subseteq M$ and disorders $D_I \subseteq D$, where $\mathit{effects}(D_I) = \bigcup_{d_i \in D_I} \mathit{effects}(d_i)$ and $\mathit{causes}(M_J) = \bigcup_{m_j \in M_j} \mathit{causes}(m_j)$. Finally, D_I is a *cover* of M_J if $M_J \subseteq \mathit{effects}(D_I)$, that is, a set of disorders covers a set of manifestations if the set of manifestations is a subset of the set of effects of the disorders.

The question then becomes, how can we determine which covers are more plausible than others? Parsimonious covering theory suggests that simpler covers are more likely to be true than complex ones. This follows *Occam's Razor*, which states that the simplest solution is most likely to be the correct one [4]. It is then these simple, or *parsimonious*, covers, that we wish to find when diagnosing a problem. The definition of what exactly constitutes a simple cover, however, is not so cut and dry. There are several different suggested criteria for judging the simplicity of a cover.

A *single-disorder* cover is a cover D_I of M_J that consists of only one disorder. A *minimal* cover is a cover D_I of M_J that contains the minimal number of disorders required to cover M_J. This may, of course, turn out to be a single-disorder cover, but it is not necessarily one. An *irredundant* cover is a cover D_I of M_J where each disorder causes at least one manifestation that no other disorder in the cover causes. There are no disorders in D_I that are not needed to cover M_J. In more formal terms, there is no $D_K \subset D_I | D_K$ is a cover of M_J. A *relevant* cover is a cover D_I of M_J that contains no disorders that are not a cause of at least one manifestation in M_J. 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.

Let us return to the automotive example from [23] that we used in Section 3.1 to illustrate the various parsimony criteria. Say that the set of present manifestations, M^+, contains *left headlight does not come on* and *right headlight does not come on.* We can see logically that any cover that contains *battery dead* or both *left headlight burned out* and *right headlight burned out* will cover our manifestations, giving us a total of 10 covers of M^+. This set of covers includes a cover containing all 4 disorders, including the *fuel line blocked* disorder that has no causal association to either manifestation in M^+. Clearly this cover should be excluded from our set of likely explanations, and as such we will move to the set of Relevant covers. Any disorder in a relevant cover must explain at least one manifestation in M^+, thus eliminating *fuel line blocked* from consideration, and leaving us with a reduced set of 5 covers. This set of covers includes a cover containing all three of the remaining disorders, that is, *battery dead, left headlight burned out* and *right headlight burned out.* It logically seems unlikely that all three of these disorders are present at the same time, or at least far less likely than covers containing one or two of them instead. To further reduce the set of covers, we will now accept only irredundant covers. This means that each disorder in the cover needs to explain a manifestation that is not explained by any other disorder in the cover, and brings the set of covers down to two. The two remaining covers are {*battery is dead*} and {*left headlight burned out* and *right headlight burned out*}. This seems like a reasonable set of hypothesis covers, but we will go further for the purposes of illustrating the final two parsimony conditions. The set of Minimal covers contains only covers that explain M^+ with the fewest number of disorders possible. This reduces our set of covers to only *battery is dead*, since all other covers contain more disorders than this one. The single-disorder parsimony condition requires that each cover contain only one disorder, and in our example, leaves us again with only *battery is dead.*

3.1.2 Diagnosis Algorithm

Diagnosis can be performed on a problem as defined in section 3.1.1 to build a set of hypotheses to explain observed manifestations. An algorithm has been developed by Peng and Reggia in [23] to diagnose the problem by constructing the set of all irredundant covers of the present manifestations. Each cover represents a single hypothesis solution, giving one potential explanation for the manifestations. The algorithm constructs the hypothesis set by including each manifestation one at at time. It begins with the first manifestation, where the hypotheses are simply the causes of the manifestation, and then proceeds to update the hypothesis set to cover each subsequent manifestation until all are included.

Generators

Within the algorithm, sets of hypotheses (covering disorder sets) are represented by a structure called a *Generator.* Many disorder sets can be represented by a single generator. A generator is simply a set of disjoint disorder sets. The generator represents all disorder sets that can be built by taking one disorder from each disorder set in the generator. Formally, if $g_1, g_2, \ldots, g_n$ are non-empty disjoint sets and $g_i \subseteq D\ \forall 1 \leq i \leq n$, then $G_I = \{g_1, g_2, \ldots, g_n\}$ is a *generator.* The set of disorder sets, or the *class* generated by generator G_I, is defined as $[G_I] = \{\{d_1, d_2, \ldots, d_n\} | d_i \in g_i, 1 \leq i \leq n\}$ [23]. A single generator cannot represent all possible disorder sets alone, and as such, a *generator set* is used to represent all disorder sets in a diagnosis. A generator set is defined as $G = \{G_1, G_2, \ldots, G_N\}$, with the sets of disorder sets generated by each generator G_I being disjoint from each other. That is, for each generator $G_I \in G$, $[G_I] \cap [G_J] = \emptyset, \forall I \neq J$. The class generated by G is $[G] = \cup_{I=1}^{N} [G_I]$ [23]. For ease of reading, we denote a generator as $G_I = (g_1, g_2, \ldots, g_n)$ instead of $G_I = \{g_1, g_2, \ldots, g_n\}$.

Say we have a generator set $G = \{G_1, G_2\}$, where $G_1 = (\{d_1, d_2\}, \{d_3\})$ and $G_2 = (\{d_4\}, \{d_5, d_6, d_3\})$. Generator G_1 would produce the disorder sets $\{d_1, d_3\}$ and $\{d_2, d_3\}$, while G_2 would produce $\{d_4, d_5\}$, $\{d_4, d_6\}$ and $\{d_4, d_3\}$. Note that a generator $G_3 = (\{d_1, d_2\}, \{d_3, d_4, d_5\}, \{d_6, d_7\})$ actually represents 12 different disorder sets.

A generator set containing all of the irredundant covers from our example in Section 3.1.1 would be $G = \{(\{\text{battery dead}\}), (\{\text{left headlight burned out}\}, \{\text{right headlight burned out}\})\}$. This generator set contains two very simple generators. The first contains only a single disorder set, containing a single disorder, generating the single hypothesis *battery dead.* The second contains two disorder sets that again contain only a single disorder each, thus generating a single hypothesis containing both *left headlight burned out* and *right headlight burned out.* To demonstrate a slightly more complicated generator, let us build a generator that represents all relevant covers of the manifestation set M^+, simply for illustration purposes. Note that this would not be done normally, as the diagnosis algorithm presented shortly is designed to produce irredundant covers. This generator set would be $G = \{(\{\text{battery dead}\}), (\{\text{left headlight burned out}\}, \{\text{right headlight burned out}\}), (\{\text{battery dead}\}, \{\text{left headlight burned out, right headlight burned out}\}), (\{\text{battery dead}\}, \{\text{left headlight burned out}\}, \{\text{right headlight burned out}\})\}$. The first two generators in the set are the same as in our irredundant cover generator set. The third generates two covers, one containing *battery dead* and *left headlight burned out* and the other containing *battery dead* and *right headlight burned out.* Finally, the fourth generator builds a single cover contain *battery dead, left headlight burned out* and *right headlight burned out.*

Generator Operations

Now that we have generators and generator sets to represent our hypothesis disorder sets, we can define a set of operations over generators that will be useful in the diagnosis algorithm. These operations will allow us to update a hypothesis generator set to cover additional manifestations. Let us say that $G = \{G_1, G_2\}$ is a generator set derived from the diagnosis of manifestations M_+, which contains one or more manifestations. That is, G represents all irredundant covers of M_+. We want to add a new manifestation into M_+ and update our diagnosis, G. To do this, we will make use of the following operations over G, along with the set of causes of the new manifestation. We designate the set of causes as the disorder set H_1.

The *division* of generator set G by disorder set H_1, $div(G, H_1)$, results in a new generator set that contains all of the covers within G that are also irredundant covers of the new manifestation. The division operation can also be extended to calculate the division of G by another generator or generator set rather than simply by a single disorder set.

The *residual* of the division of G by H_1, $res(G, H_1)$, then results in a new generator set that contains all of the covers within G that are not irredundant covers of the new manifestation. As with division, the residual operation can also be extended to calculate the residual of G divided by another generator set rather than simply by a single disorder set.

The *augmented residual* of G divided by H_1, $augres(G, H_1)$, results in a new generator set that contains the results of the residual of G divided by H_1 modified to also be irredundant covers of the new manifestation. This is done by simply adding H_1 to each generator set in $res(G, H_1)$.

Finally, the *revise* operation, $rev(G, H_1)$, results in a new generator set that contains all irredundant covers of M_+ with the new manifestation, and it does so by making use of all three of the other operations. Formally, $revise(G, H_1) = F \cup res(Q, F)$, where $F = div(G, H_1)$ and $Q = augres(G, H_1)$. Essentially, the revise operation needs to keep all disorder sets in G that are also irredundant covers of the new manifestation, and modify the rest so that they are. This is done by calculating the division of G by H_1 and the augmented residual of that division, as is done with $F = div(G, H_1)$ and $Q = augres(G, H_1)$. The use of the residual of Q and F is done to remove duplicates and redundant covers from Q [23].

The formal definitions of these operations can be found in [23].

Algorithm

The actual diagnosis algorithm is quite simple. It makes use of the causal network, generators, and the functions and operations defined to use and manipulate them. The algorithm starts with an empty generator set to store the hypothesis, and is given the set of current manifestations. It then iterates through each manifestation, revising the hypothesis generator 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.

Let us make a final return to our basic example of automotive diagnosis from [23], illustrated in Figure 3.1, and take a high level look at the diagnosis algorithm in action. Let the set of current manifestations, M^+, contain *engine does not start, left headlight does not come on*, and *right headlight does not come on.* We start with an empty hypothesis generator set $G = \{\emptyset\}$, and proceed to loop through each manifestation in M^+, one at a time. The first manifestation is *engine does not start.* We revise our current hypothesis, which is empty, by including the causes of

```
1: generatorSet hypothesis = $\{\emptyset\}$
2: while moreManifestations do
3:     $m_{new}$ = nextManifestation;
4:     hypothesis = revise(hypothesis, causes($m_{new}$))
5: end while
6: return hypothesis
```

Figure 3.2: Diagnosis Algorithm

engine does not start, which are *battery dead* and *fuel line blocked.* This results in a new generator set $G = \{(\{\text{battery dead, fuel line blocked}\})\}$, indicating that our possibilities are that either the battery is dead, or the fuel line is blocked. Next we add the *left headlight does not come on* manifestation, and revise the hypothesis set with it's causes (*battery dead* and *left headlight burned out*). This brings us to a new generator set $G = \{(\{\text{battery dead}\}), (\{\text{fuel line blocked}\}, \{\text{left headlight burned out}\})\}$. In this set, the first generator indicates that the battery is dead and explains both manifestations. The second generator builds a cover that uses two disorders to explain the manifestations, *fuel line blocked* and *left headlight burned out*, to explain *engine does not start* and *left headlight does not come on*, respectively. Finally, we add the last manifestation, *right headlight does not come on*, using its causes (*battery dead* and *right headlight burned out*). Revising the hypothesis set using these causes brings us to our final hypothesis generator set, $G = \{(\{\text{battery dead}\}), (\{\text{fuel line blocked}\}, \{\text{left headlight burned out}\}, \{\text{right headlight burned out}\})\}$. This generator set constructs two hypotheses. The first contains only *battery dead*, and the second contains *fuel line blocked*, *left headlight burned out* and *right headlight burned out.* Both of these are irredundant covers of M^+. 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.

Chapter 4

Implementation

A version of the Abductive Diagnosis algorithm described in Section 3.1.2 was implemented in the BEAT autonomic manager, described in Section 2.4.1. The diagnosis algorithm replaces the existing method of action selection in BEAT. The BEAT autonomic manager and all of the modifications described in Section 4.2 are written in the C++ programming language.

4.1 Abductive Diagnosis Implementation

The Abductive Diagnosis data and algorithm were modeled and built first, separate from the BEAT system. This was done in an effort to ensure that the code was not coupled to the specifics of the BEAT implementation, but would rather be general enough to port to future work.

4.1.1 Causal Network

A set of classes were created to model the causal relationship knowledge and perform the diagnosis described in Section 3.1.2. Figure 4.1 shows this set of classes with a simplified set of properties and methods. The CausalNetwork class represents the entire causal network, which consists of a set of manifestations and and a set of disorders, represented by the Manifestation and Disorder classes, respectively. The Manifestation class is associated with one or more Disorder classes, indicating that

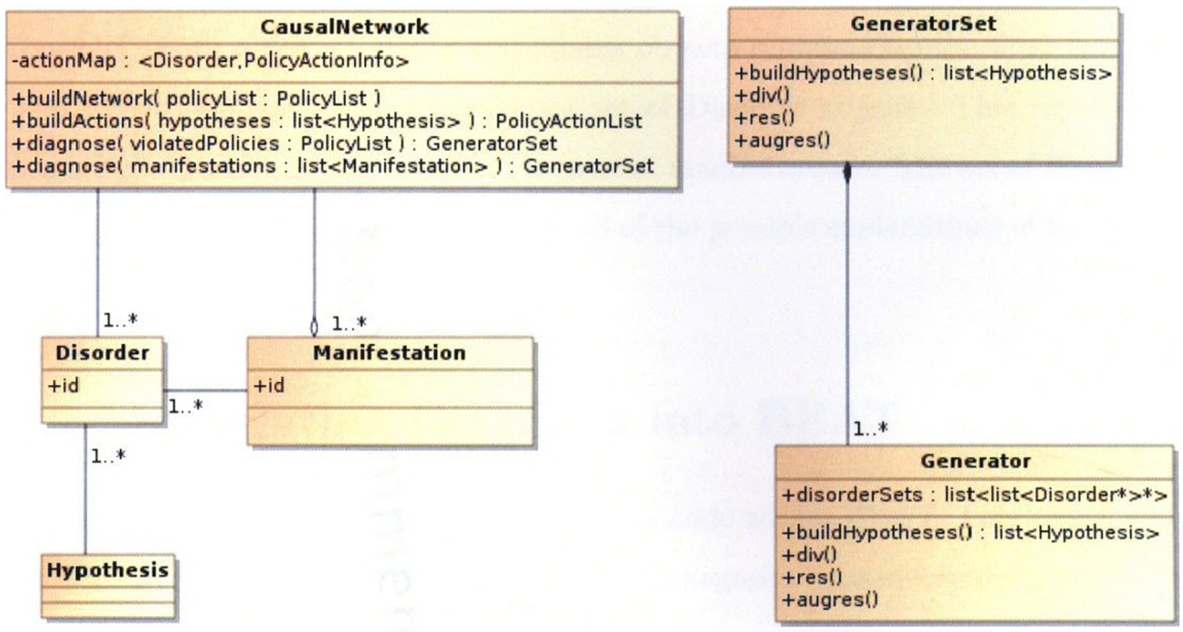
CausalNetwork
-actionMap : <Disorder,PolicyActionInfo>
+buildNetwork(policyList : PolicyList)
+buildActions(hypotheses : list<Hypothesis>) : PolicyActionList
+diagnose(violatedPolicies : PolicyList) : GeneratorSet
+diagnose(manifestations : list<Manifestation>) : GeneratorSet
GeneratorSet
+buildHypotheses() : list<Hypothesis>
+div()
+res()
+augres()
Disorder
+id
Manifestation
+id
Hypothesis
Generator
+disorderSets : list<list<Disorder*>*>
+buildHypotheses() : list<Hypothesis>
+div()
+res()
+augres()
1..*
1..*
1..*
1..*
1..*

Figure 4.1: Causal Network Class Diagram

in the causal network, the manifestation is known to be caused by the associated disorders. The CausalNetwork class has methods to build the network out of a set of policies, which will be discussed in Section 4.2, to perform a diagnosis given a set of present manifestations (a list of Manifestation objects), and build a set of actions from a list of hypotheses, which will also be discussed in Section 4.2.

4.1.2 Diagnosis Algorithm

The diagnosis algorithm is coded as described in Section 3.1.2. The main algorithm is run within the diagnose() method of the CausalNetwork class. Generators and generator sets are represented by the Generator and GeneratorSet classes, respectively, as seen in Figure 4.1. The Generator set contains a set of disorder sets, and the GeneratorSet class contains a set of Generator objects, exactly as indicated in the description of the algorithm in Section 3.1.2. These classes contain methods to perform the generator operations required by the diagnosis algorithm, that is, the division, residual, and augmented residual operations. The diagnose() method of

CausalNetwork returns a list of Hypothesis objects representing the final diagnosis. The Hypothesis class simply contains a set of Disorder objects. This represents a single hypothesis as to the cause of the present manifestations. The set of Hypothesis objects returned by diagnose() represent all of the possible explanations of the present manifestations.

4.2 Integrating Diagnosis into BEAT

The next step was to implement the diagnosis code within BEAT. This was done by both modifying BEAT and adding methods to diagnosis classes to aide in converting between BEAT policy classes and diagnosis classes.

The BEAT autonomic manager required modification in two main locations to add diagnosis, namely, the Policy Decision Point (PDP) and the Policy Enforcement Point (PEP).

4.2.1 Policy Decision Point

The Policy Decision Point, as described in Section 2.4.1, detects policy violations (based on received events) and decides which action should be taken. An ordered list of potential actions is sent to the Policy Enforcement Point, where the first action to pass its associated tests (see Section 2.4.1) is executed.

Causal Network Construction

The first step in integrating diagnosis in the BEAT system is to construct the Causal Network that will contain the causal relationship knowledge used to perform the diagnosis. The Causal Network can be constructed at run-time using the policy information already provided to the system. 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 object. 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 not to be set correctly 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 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 Manifestation and Disorder classes 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. This derivation of the causal network from policies is done by the buildNetwork() method of the CausalNetwork class, which accepts a list of all of the policies in the system as input. Diagnosis performed on a causal network built in this manner then essentially finds actions or sets of actions that can potentially cause all present conditions to no longer be true, thus eliminating all policy violations.

Performing the Diagnosis

Diagnosis is performed by calling the diagnose() method of the CausalNetwork class. This method accepts a list of policies, representing all policies that are currently in violation. Within the method, the conditions of each policy are extracted and used to look up the Manifestation classes built from them. A second version of the diagnose() method that accepts a list of Manifestation classes representing present manifestations is then called, which returns a set of hypotheses (a list of Hypothesis objects).

Ordering Hypotheses

Since the Policy Decision Point needs to send an ordered list of actions to the Policy Enforcement Point for execution, hypotheses need to be ordered in some manner. This ordering is essentially a ranking, as the PEP will execute the first one it receives that passes it's associated tests. 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.

Converting Hypotheses to Actions

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, which is stored in the actionMap property of the CausalNetwork class, as seen in Figure 4.1. The actions for a single hypotheses 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.

Executing Actions

Modifications were made to the format of the messages passed to the Policy Enforcement Point to allow for multiple actions to be executed simultaneously. Since a hypothesis may contain more than one diagnosis, it may be necessary to perform more than one action at a time, which was not possible in the original BEAT implementation. Actions should be sent in groups of one or more to the PEP for execution.

Section 4.2.2 below describes modifications made to the PEP to make this possible.

4.2.2 Policy Enforcement Point

The Policy Enforcement Point is responsible for executing policy actions, as described in Section 2.4.1. It receives an ordered list of actions from the PDP and executes the first action that passes its associated tests. In the original implementation, the PEP was capable of performing only a single action at a time. Since the diagnosis algorithm may indicate that several simultaneous actions are required to cover all of the present policy violations, the PEP was modified to include the ability to execute multiple actions. Instead of a list of single actions, the PEP now accepts a list of action groups, with each group containing one or more actions. The associated tests are performed on each individual action, and the entire group executes only if every action within it passes. Again, the first group that passes its tests is executed in its entirety.

Chapter 5 Experiments

In order to make some judgment on the performance of the implemented diagnosis algorithm, we need to compare it to other methods of selecting policy actions. We do this by configuring the BEAT autonomic manager [14] 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, with the newly developed diagnosis algorithm, as well as 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.

5.1 Test Environment

The test environment consists of a web server running the BEAT autonomic manager and several client machines generating load for the server by requesting web pages over HTTP. All machines reside on the same LAN, connected via wired Ethernet using a D-Link DIR-655 Gigabit Router.

5.1.1 Test Systems

Differentiated Services

Differentiated Services (DiffServ) refers to applying different quality of service rules to traffic from different sources [16]. Monitors and Effectors included with BEAT provide the ability to distinguish three service classes: gold, silver and bronze. Incoming requests are assigned to one of these classes based on the source of the request. In a real-life application, this may be the result of users paying more money for a higher level of guaranteed service. As resources are consumed, DiffServ is a tool that can be used by policies within BEAT to compromise the service of lower classes in order to maintain quality of service for higher classes. The gold class has the highest ranking and receives top priority, with silver being second and bronze third.

Server

The server machine hosts a web server running a PHP bulletin board application [6]. The application makes use of a database, also running on the server machine. The server is a LAMP stack (Linux, Apache, MySQL, and PHP), and consists of:

- Fedora 11 Operating System [2]
- Apache 2.0 HTTP Server [8]
- MySQL 5.1 Database [5]
- PHP 5.2.9 [7]

The BEAT autonomic manager is installed on the server machine to provide policy-based autonomic management. See Section 2.4.1 for more details on BEAT.

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 given highest priority, and requests sent by the bronze machine are 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 arbitrary prioritization scheme. The client machines consist of:

- Fedora 11 Operating System [2]
- Apache JMeter 2.3.4 Load Generator [1]

5.1.2 Load Generation

Creation of the workload for the server is done on each client machine through the use of Apache JMeter [1]. JMeter is an open-source, Java based load generator, capable of generating loads for HTTP server, database, web services, and more. It provides a flexible means of designing tests and extensive logging capabilities to record results. It has multithreading support, allowing each thread to represent a 'user' sending requests to the server [1]. A single instance of the JMeter GUI can be used to control several instances of JMeter on separate machines, combining the results into a single log [1]. Apache JMeter has been used exclusively for load generation in these experiments.

5.2 Systems Under Test

Four configurations will be contrasted with each other to determine 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 2.3 outlines a set of criteria that can be used to guide action selection. These criteria, (severity, specificity, weight and advocacy), are implemented in BEAT [12] 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. [12]. 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 variance 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.

5.3 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

The 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 services classes to analyze the effects of service differentiation.

Throughput

The throughput of the server is calculated as the number of requests it can complete per second. This will be based on data collected from the client machines.

5.4 Policy Goals

The set of policies in use define what the autonomic manager should do in specific cases. These 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, which are designed to measure metrics against goal threshold values. 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.

For more details on the metrics being measured, see Section 5.3.

5.5 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. The experiment was repeated a total of 5 times to reasonably ensure that the results are reproducible and not due to any unrelated anomaly.

5.5.1 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.

	Disabled	Weighted	Diagnosis Fewer	Diagnosis Many
Apache Resp.	3336ms [1183]	1195ms [898]	1031ms [843]	1163ms [913]
CPU Util.	98.3% [10.7]	74.4% [18.4]	82.5% [18.3]	82.4% [19.0]
Memory Util.	22.3% [1.2]	24.6% [2.2]	24.0% [1.9]	24.1% [2.1]

Table 5.1: Average Server Results

5.5.2 Measurements

Averages, standard deviations, and throughput values were calculated for each run and averaged over the 5 repeats of the experiment. We break the results down into two sections, results measured from the server and results measured from the client machines, to make them more easily digestible.

Server Metrics

Table 5.1 shows the metrics measured by the server machine. These include the response time of the Apache web server (as measured by the mechanism described in Section 5.3), CPU Utilization and Memory Utilization. These are the most relevant measures to the evaluation of the diagnosis algorithm as compared to weighted action selection, since this is the same information available to the autonomic manager at runtime to trigger policy decisions. The values shown are the average values for an entire run, with the standard deviation shown in square brackets. The metric averages and standard deviations are then averaged across all 5 replications of the experiment. Figure 5.1 shows the Apache Response Time data from table 5.1 as a box plot, and figure 5.2 shows the CPU Utilization data as a box plot.

Figure 5.2 shows the same metrics as figure 5.1, 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 period when the server was operating under maximum load (75 clients total). The results are slightly

	Disabled	Weighted	Diagnosis Fewer	Diagnosis Many
Apache Resp.	3722ms [587]	1300ms [883]	1095ms [822]	1247ms [875]
CPU Util.	99.9% [2.3]	78.0% [10.4]	86.8% [7.8]	87.2% [8.9]

Table 5.2: Average Server Results - Max Load Only

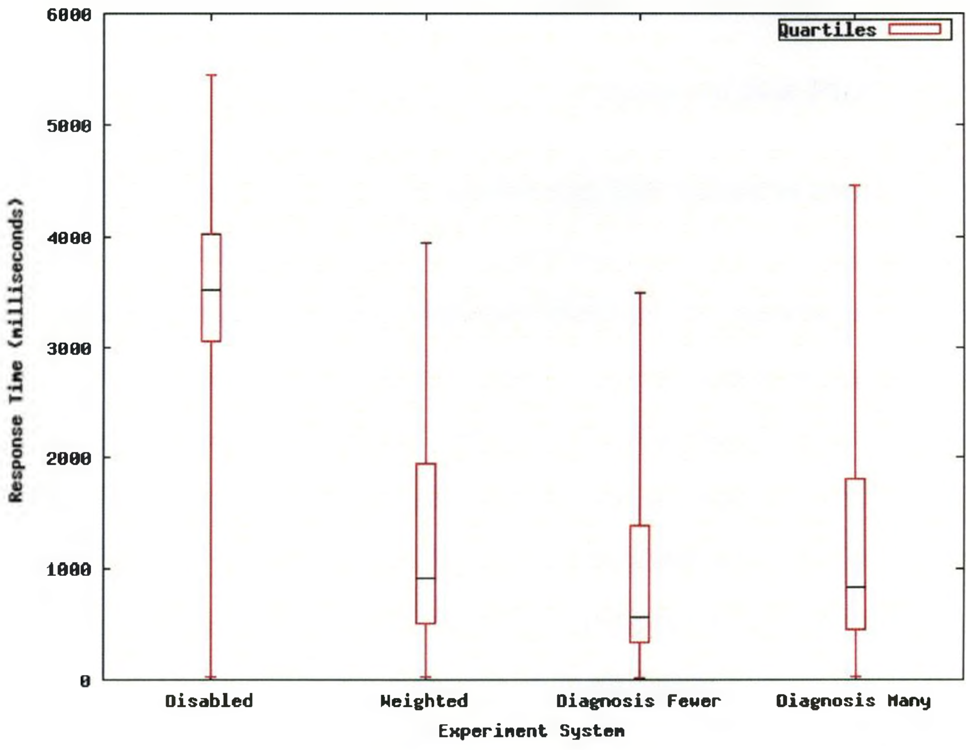

Figure 5.1: Apache Response Time Box Plot

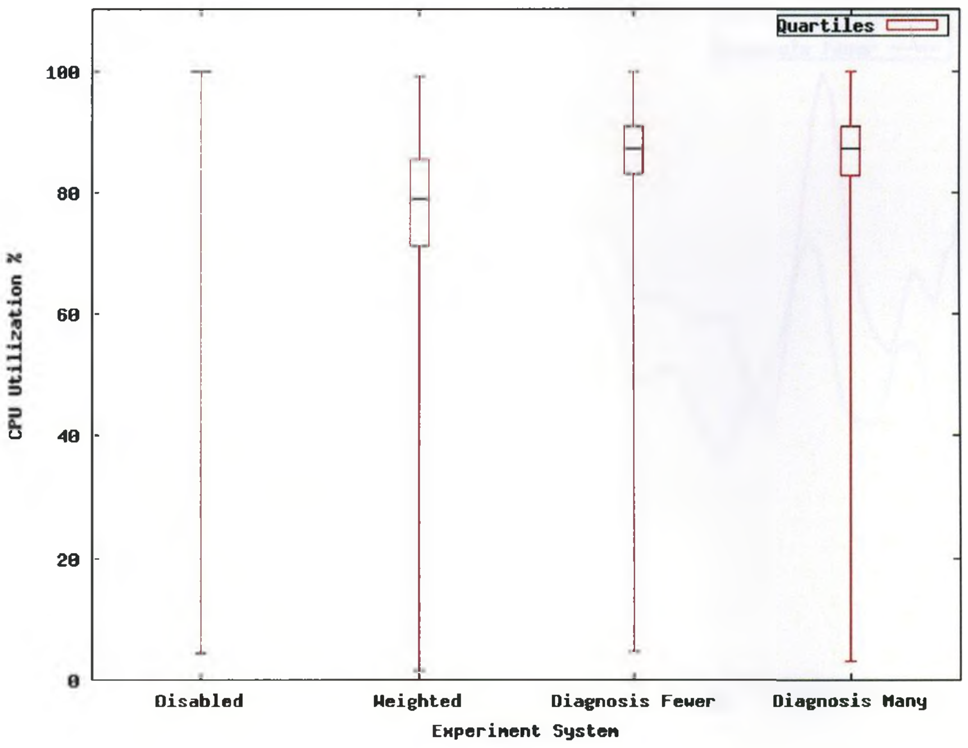

Figure 5.2: CPU Utilization Box Plot

different in value to those of the entire run, but values in comparison with each other remain consistent.

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 similar, with Diagnosis favouring hypotheses with fewer disorders (fewer actions to take) beating out the other two, which match up fairly evenly. Figure 5.3 compares the Apache response times for weighted action selection and diagnosis favouring fewer actions. 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

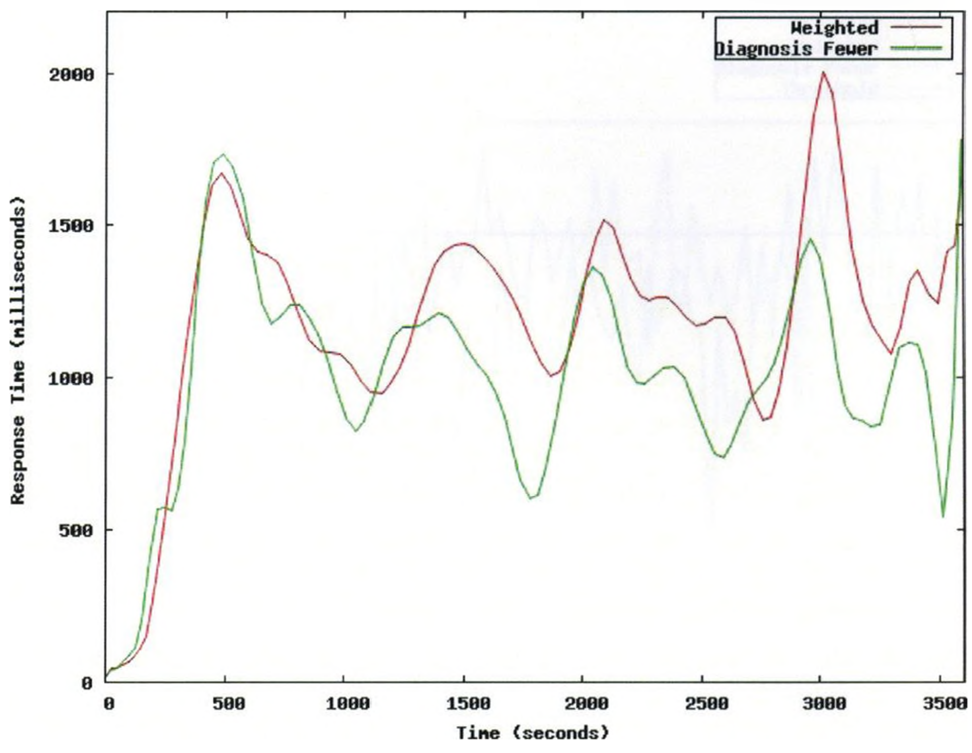

Figure 5.3: Apache Response Time

smooth the data. The data shown is from a single experiment, not averaged over all 5 repetitions, and represents results consistent with all experiments.

Both forms of the diagnosis algorithm make better use of the CPU, as outlined in the goals of our set of policies in Section 5.4, without going over the 90% threshold. Diagnosis favouring hypotheses with many disorders (more actions to perform), however, doesn't see any improvement over the response time of weighted action selection. Figure 5.4 compares the CPU utilization of the system with the manager disabled, with weighted action selection, and with diagnosis favouring fewer actions. Again, the data shown is from a single experiment, not averaged over all 5 repetitions, and represents results consistent with all experiments.

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 5.5 shows the measured

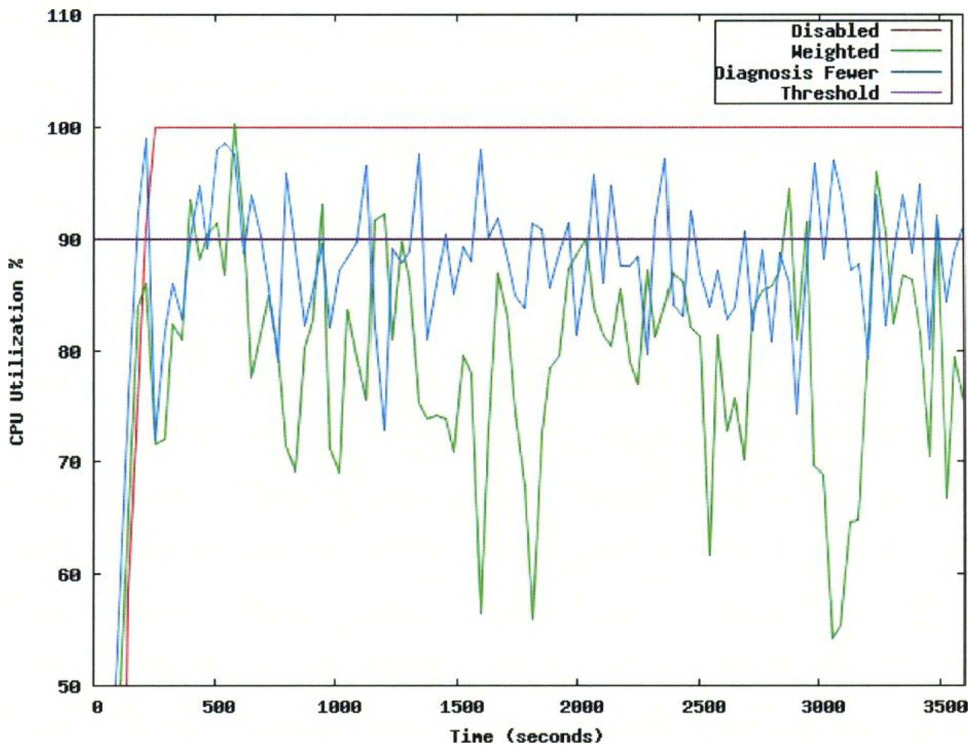

Figure 5.4: CPU Utilization

response time of the Apache web server with the manager disabled and with weighted action selection, compared to the threshold value of 2000ms. 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 5.3 contains these values. The values shown are averaged over the 5 experiment repetitions, with standard deviations for the repetitions shown in square brackets. Note that this differs from figure 5.1, where the standard deviations are for the individual metric values, and are themselves averaged across the 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. CPU utilization values over the threshold are negligible for all three methods of action selection. Diagnosis favouring fewer actions comes out on top yet again, with weighted actions and diagnosis features multiple actions coming in second and third, respectively.

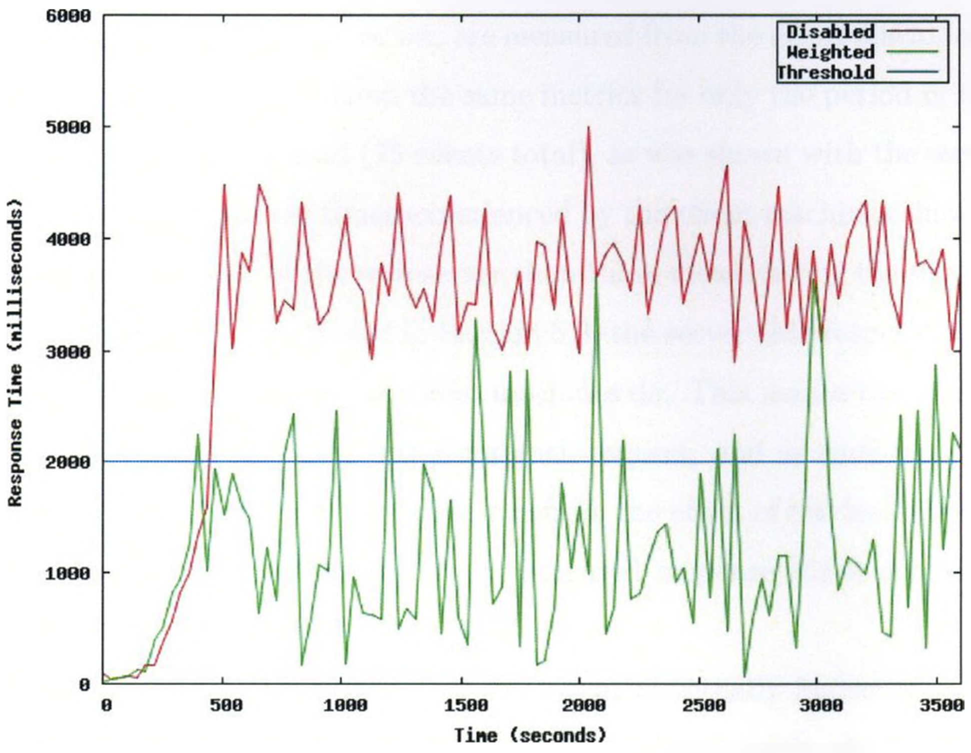

Figure 5.5: Apache Response Time Over Threshold

	Disabled	Weighted	Diagnosis Fewer	Diagnosis Many
Apache Resp.	290.30 [17.61]	14.94 [2.82]	11.19 [2.55]	16.19 [6.34]
CPU Util.	3.65 [0.03]	0.04 [0.01]	0.13 [0.02]	0.15 [0.02]

Table 5.3: Server Metrics Area Over Threshold

Client-side Metrics

The actual response times experienced by the three service classes, as well as overall throughput values for the server, are measured from the client machines, and are listed in Table 5.4. Table 5.5 shows the same metrics for only the period in which the server was under maximum load (75 clients total), as was shown with the server-side metrics in figure 5.2. 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 5.3, 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. As such, the client measured response times can be quite different from the server measured response time.

Throughput as measured from the client is actually higher when the autonomic manager is disabled. This is a logical and expected result, though, as CPU usage is significantly higher due to the restrictions placed on CPU usage by the policies when the manager is enabled (must be below 90%). A trade-off is made between response time and CPU usage, as the autonomic manager policies attempt to achieve not only the response time goal, but the CPU and Memory goals as well. Throughput for the two diagnosis methods is slightly better than weighted action selection, which matches up with the CPU usage seen in Table 5.1.

The most important of the response time measures is that of the gold service class. 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 5.6 is a box plot of the Gold Class Response Time data used in table 5.4. 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 5.7 shows the gold, silver, and bronze response times for the

	Disabled	Weighted	Diagnosis Fewer	Diagnosis Many
Gold Resp. Avg.	2182ms [923]	1798ms [1824]	1389ms [675]	1465ms [741]
Silver Resp. Avg.	2228ms [931]	4021ms [5165]	3920ms [4284]	3827ms [3966]
Bronze Resp. Avg.	2192ms [912]	4742ms [8072]	5543ms [8572]	5239ms [7477]
Throughput	21.8/s	17.0/s	18.1/s	18.1/s

Table 5.4: Client Results

	Disabled	Weighted	Diagnosis Fewer	Diagnosis Many
Gold Resp. Avg.	2333ms [833]	1872ms [1892]	1398ms [644]	1463ms [689]
Silver Resp. Avg.	2365ms [834]	4442ms [5535]	4387ms [4532]	4257ms [4199]
Bronze Resp. Avg.	2336ms [824]	5404ms [8863]	6657ms [9461]	6233ms [8217]
Throughput	18.6/s	14.3/s	15.2/s	15.3/s

Table 5.5: Client Results - Max Load Only

system using the diagnosis algorithm favouring fewer actions. Service differentiation is clearly visible in this graph, as the system keeps the gold class consistent at the expense of silver and bronze.

5.6 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, and 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

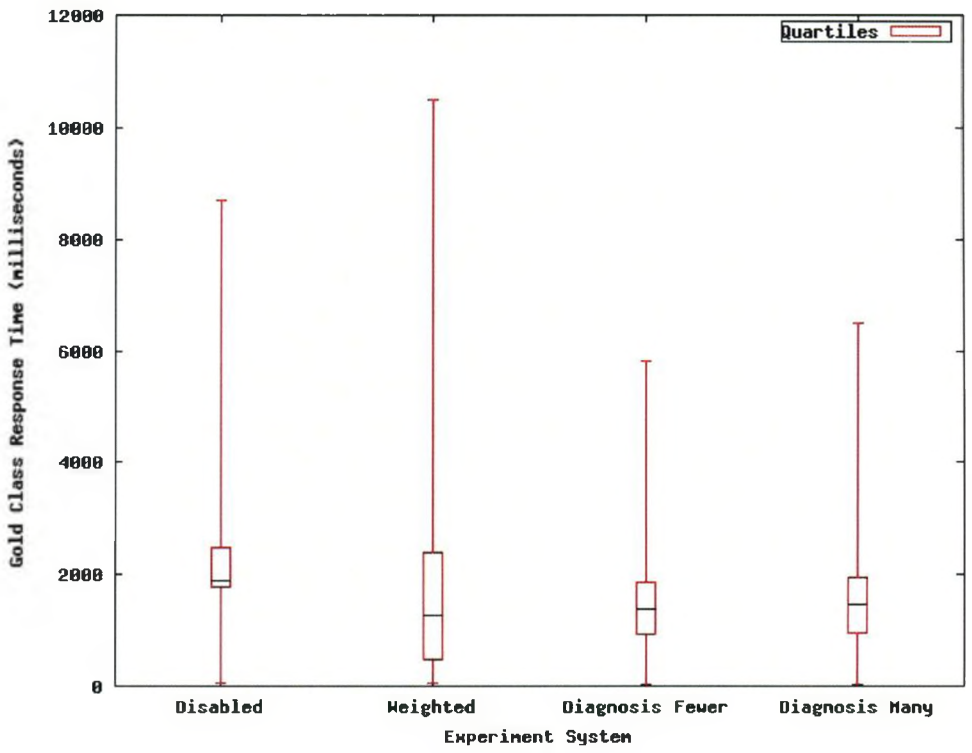

Figure 5.6: Gold Response Time Box Plot

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 5.8 and 5.9 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

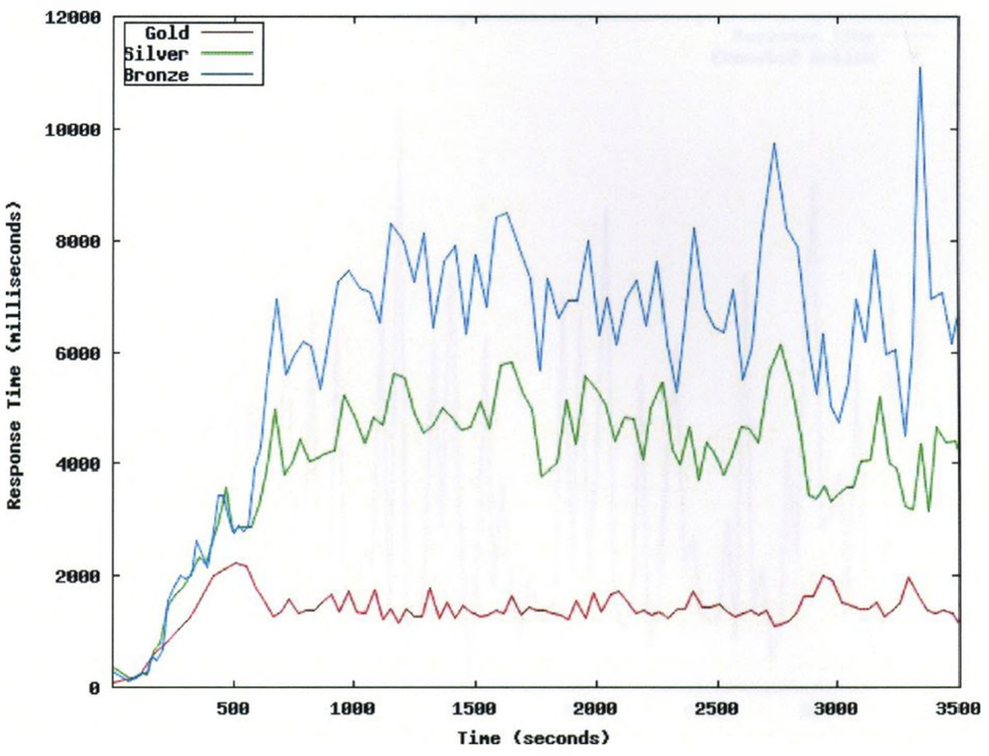

Figure 5.7: Client Response Times for Diagnosis Fewer

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

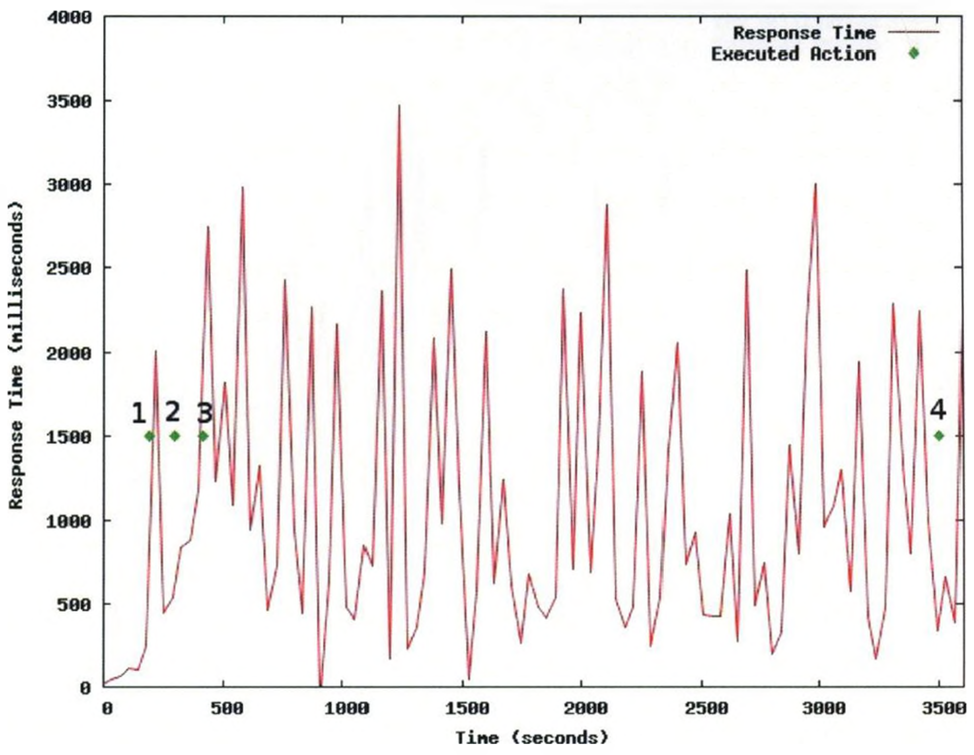

Figure 5.8: Example Run with Diagnosis - Response Time

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

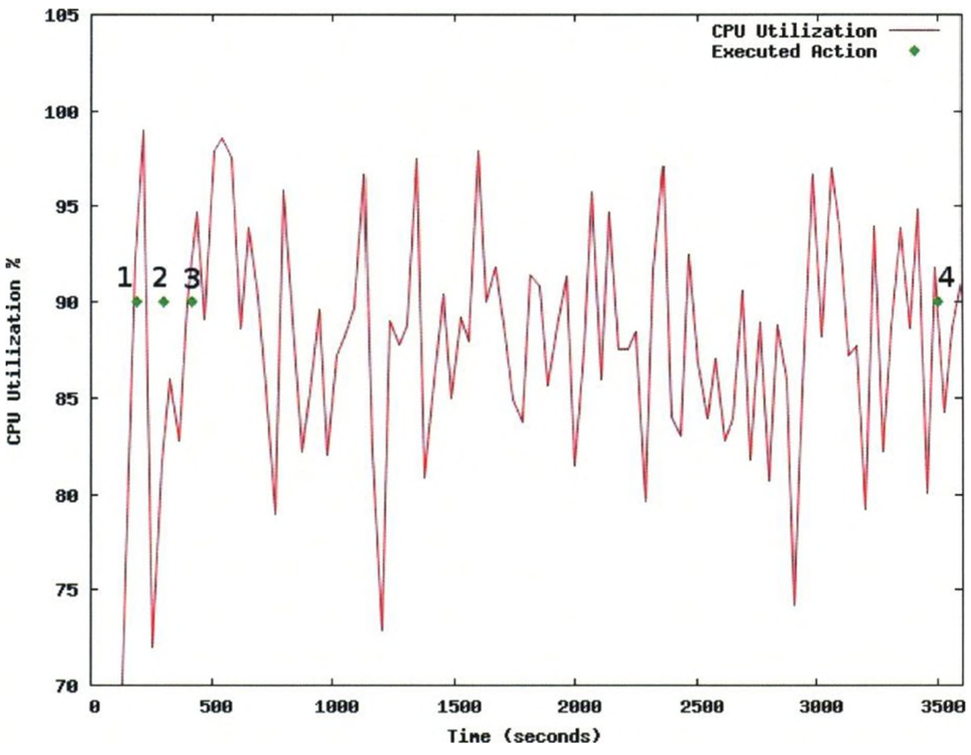

Figure 5.9: Example Run with Diagnosis - CPU Utilization

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 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). Tests for actions are described in Section 2.4.1. 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.

5.7 Discussion

We have examined the performance of a web server without the aid of the BEAT autonomic manager, with the manager using weighted action selection, using diagnosis favouring fewer actions, and with diagnosis favouring multiple actions. 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. 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. 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 [4]. 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.

Chapter 6
Conclusions and Future Work

6.1 Conclusions

Policy-based Autonomic Management attempts to dynamically manage, configure and optimize a set of running applications in real-time. A set of policies form the knowledge of the system, with a policy *violation* requiring some action to be taken to correct it. In the common case of multiple policy violations, there are often many potential actions that can be taken to correct them. Deciding which action will be the most effective is a difficult decision, with a high impact on how well the autonomic manager will achieve the goals of the policies.

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 using specific sets of violations. The diagnosis algorithm was implemented in the BEAT Autonomic Manager [14].

The performance of the diagnosis method of action selection was compared to that of another, previously developed method. This previous method, which we call *weighted action selection*, uses a number of factors to assign weights to actions, and executes the action with the highest weight. Two versions of the diagnosis method, one favouring the execution of the least number of actions possible and the other favouring the most, were implemented. The BEAT autonomic manager was configured to manage a basic server machine running both a web server and a database. The performance of the server under load was measured without the autonomic manager, with the autonomic manager using weighted action selection, and with the autonomic manager using the two different versions of diagnosis.

The results look promising, with diagnosis favouring fewer actions outperforming the other methods. It seems to make better decisions on which action or actions to perform, more closely achieving the overall goals of the policies, that is, keeping metrics such as CPU and Response Time within specified thresholds.

In closing, a new method for selecting actions to perform based on multiple policy violations was developed. An abductive diagnosis algorithm was implemented, including the development of a method of building a causal network from a set of policies. Overall, the diagnosis algorithm performed well in our experiments, with diagnosis favouring fewer actions slightly outperforming the previously implemented method of action selection (weighted action selection) in the BEAT Autonomic Manager.

6.2 Future Work

There is a good deal of room for improvement in both action selection as a whole and the diagnosis approach itself. The diagnosis method is not a strict alternative to weighted action selection, and in fact these methods could be easily combined. The criteria for policy and action weighting could be used to build probabilities into the causal network, which could be used to help order hypotheses returned from the diagnosis algorithm. This may lead to even further performance improvements.

The policies themselves could also be examined. Some policies are designed to combine the conditions of more than one policy violation in order to take into account as much of the information as it can. Clearly this can lead to an unmanageable number of policies to design as the manager handles both more applications and more aspects of each application. Diagnosis inherently makes use of all of the existing policy violation information in its decision making, thus rendering such additional policies potentially obsolete. This could allow for the development of simpler sets of policies.

Finally, in order to fully evaluate and drive development of these techniques forward, some larger scale implementation and testing is likely necessary. This would provide a better idea as to how the the diagnosis and other methods would hold up where there are needed most, in a more realistic, large scale deployment.

Bibliography

[1] "Apache JMeter," http://jakarta.apache.org/jmeter/, August 2009.

[2] "Fedora 11," http://fedoraproject.org/, August 2009.

[3] "Merriam-Webster Online Dictionary, Diagnosis entry," http://www.merriam-webster.com/dictionary/diagnosis, May 2009.

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

[5] "MySQL Database," http://www.mysql.com/, August 2009.

[6] "PHP Bulletin Board," http://www.phpbb.com/, August 2009.

[7] "PHP: Hypertext Preprocessor," http://www.php.net/, August 2009.

[8] "The Apache HTTP Server Project," http://httpd.apache.org/, August 2009.

[9] "The Free On-line Dictionary of Computing," http://dictionary1.classic.reference.com/browse/abduction, May 2009.

[10] R. Anthony, "Policy-centric Integration and Dynamic Composition of Autonomic Computing Techniques," in *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, 2007, p. 2.

[11] R. Bahati, M. Bauer, and E. Vieira, "Mapping Policies into Autonomic Management Actions," in *Autonomic and Autonomous Systems, 2006. ICAS'06. 2006 International Conference on*, 2006, p. 38.

[12] ——, "Adaptation Strategies in Policy-Driven Autonomic Management," in *International Conference on Autonomic and Autonomous Systems.* IEEE Computer Society Washington, DC, USA, 2007, p. 16.

[13] ——, "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.

[14] 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, p. 5.

[15] 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.

[16] Cisco Systems, "DiffServ – The Scalable End-to-End Quality of Service Model," http://www.cisco.com/en/US/technologies/tk543/tk766/technologies_white_paper09186a00800a3e2f.pdf, August 2005.

[17] 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.

[18] I.B.M.A. Computing, "IBMs Perspective on the State of Information Technology," *IBM Research, Westchester County, NY*, 2001.

[19] J. Kephart, T. Center, and H. IBM, "Research Challenges of Autonomic Computing," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, 2005, pp. 15–22.

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

[21] 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.

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

[23] ——, *Abductive Inference Models for Diagnostic Problem-solving.* Springer, 1990.