RC23076 (W0401-109) January 22, 2004
Computer Science

IBM Research Report

A Toolkit for Policy Enablement in Autonomic Computing

Dinesh C. Verma, Seraphin B. Calo
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

A Toolkit for Policy Enablement in Autonomic Computing

Dinesh C. Verma
IBM Research
dverma@us.ibm.com

Seraphin B. Calo
IBM Research
scalo@us.ibm.com

Abstract

A Policy Toolkit is being developed at the IBM T.J. Watson Research Center that is aimed at accelerating the adoption of policy based technologies and methodologies. The goal is to produce a set of common software components that can be used across a wide variety of applications, and that simplify the task of integrating policy related methodologies into new or existing software systems. As such, the Policy Toolkit provides direct support for the efforts involved in autonomic computing, e-business on demand, OGSA grid computing, and web services. It contains libraries of commonly used policy manipulation functions (e.g., for creating, validating, evaluating, and otherwise managing a set of policies), as well as patterns for building typical policy-based systems. Its components can be bundled into user applications in a variety of ways, allowing them to flexibly incorporate the ability to make decisions based on policies.

1. Introduction

Developers of policy enabled systems need a common set of basic functionality regardless of their areas of application. This is most conveniently provided in terms of a toolkit from which they can choose those components that are needed for their particular implementations.

The Policy Toolkit (PTK) is written in Java and consists of a core module plus a set of modules that perform specific functions as shown in Figure 1. The toolkit is designed so that any application can select a subset of the available modules in its design. The Policy Core Classes represent policy rules, conditions, actions, etc. They provide the basic capabilities of the Policy Toolkit and are used by the other modules. The Policy Editor Module contains the classes that can be used to easily create a custom Policy Editor GUI; the Validation Module provides a set of validation checks that can be run on groups of policies; the Decomposition Module can be used to transform a high-level policy into lower-level resource specific policies; the Policy Agent Module contains the Java classes needed to easily create Policy Agent that performs the functions of policy caching and distribution; the Policy Enforcement Point Module consists of the classes that can be used to easily create a policy enforcement point for evaluating and executing the policies; and, the Policy Conflict Resolution Module has those classes that can be used to identify and resolve conflicts that may arise between groups of policies that represent different disciplines.

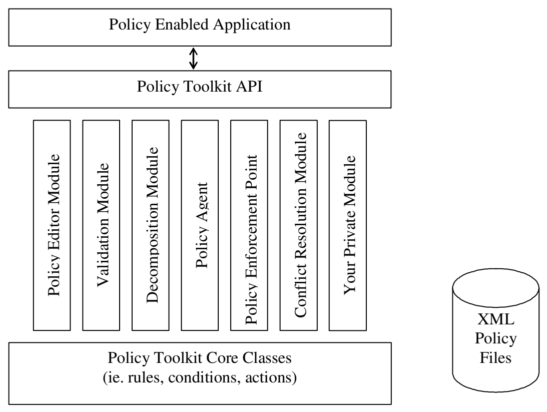

Figure 1: Policy toolkit design

The API to the toolkit is a set of java classes, and it is designed so as to support different types of policy languages as long as they conform to the policy information model described later in the paper. The default policy language supported in the toolkit is an XML representation of policies which are parsed into the java representation of the information model.

In the following sections of the paper we will present the key features of the Policy Toolkit in greater detail. Section 2 will discuss the policy model and the basic functionality provided for its support. Subsequent sections will address more advanced capabilities that have been identified as important in various engagements with those developing policy enabled applications. In Section 3, the validation of sets of policies and the runtime resolution of conflicts is considered. Section 4 deals with transformations of policies from higher levels of specification to forms that can be executed by the autonomic elements within the system. In Section 5 we present some preliminary work on policy-based design patterns; Section 6 describes an example of the use of the toolkit for policy enablement of an autonomic application; and, finally Section 7 deals with conclusions and future work.

2. Toolkit core component and design

One of the challenges in building a policy toolkit is that the policy enabled system needs to be integrated with existing system and network management consoles. As a result, it is difficult to define the notion of a policy language which will be universally acceptable. Users of existing storage management products would like to implement policies as expressed in the configuration languages of their products, as opposed to adopting a universal policy language. On the other hand, the absence of a universal language makes it difficult to provide a generic toolkit for users.

To address this problem, the PTK has been designed around policies which are implemented in accordance with a policy information model, as opposed to a specific policy language. A policy information model, e.g. the PCIMe information model [1] specifies constraints on the structure of the policy rule, without specifying the syntax for expressing the policies. A policy language, e.g. PONDER [2] is a rendering of an information model in a specific language. The toolkit has been designed to represent the constructs of an information model as a Java class, and defines an abstract interface for defining and developing parsers from a specific syntax into the information model. Different implementations of the parser interface enable the same set of Java functions to work with multiple syntactical renderings of the same policy expression.

Within the PTK, we have implemented the information model developed within the IBM Autonomic Computing Initiative. This information model is based upon the PCIMe information model, and consists of four main components – precondition, decision, business-value and scope. The precondition is a derivative of the PCIMe condition construct, the decision is a derivative of the PCIMe action construct, the business-value is a generalization of PCIMe priority values, and scope is a generalization of the PCIMe role construct. The semantics of a policy with the four components is as follows:

If the scope of a policy is applicable, and the preconditions of the policy evaluate to true, then the result of the policy must be enforced with business value used to arbitrate among multiple choices.

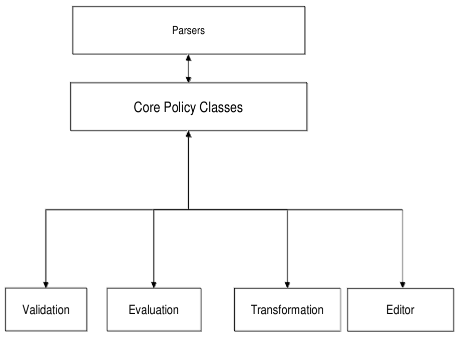

Figure 2: Policy toolkit modules relationship

Policies specified within a specific language are parsed into Java classes representing the information model, and the Java representation of the objects are used to perform the functions of the other modules within the toolkit. The different modules included within the toolkit include a policy evaluator, a policy editor system, a policy validation module, a policy transformation module, and a policy patterns module. The relationship among the different modules is shown in Figure 2. To ease the task of building policy parsers, the toolkit also includes parsers for some common types of policy languages, and some common patterns for using the toolkit to build policy enabled systems.

To build a system based on the toolkit, the system developer needs to decide on the syntax of the policy language, selects the modules for his system, and the pattern (if any) for the overall system. The developer can then customize the operation of specific modules by providing configuration information for each module (via a set of XML files) and possibly extending the interfaces provided by each of the modules.

The modules included within the policy toolkit are the following:

The core module: provides a set of basic Java classes representing policies.

The parser module: provides a way for converting policies from a syntax format to the core model. Different implementations of the module confirming to a common parser interfaces provide support for different policy languages.

Editor Module: provides a generic GUI for policy manipulation. It can be customized to specific policy syntax by using the common parser interface.

Policy agent module: provides a way to cache and receive policies from a repository.

Policy federator: provides a repository and distribution support for policies.

Validation module: The module provides for checking of consistencies among different policies.

Transformation module: It provides the ability to change policies from one format to another.

The enforcement point: provides an efficient mechanism for finding policies matching an event and finding the resulting actions.

The auto-update module: provides a way for systems to periodically update local copies of policies from a remote repository.

The patterns module: provides a set of ready-to-use templates for building policy-enabled systems.

The design of many of these modules is relatively straight-forward. Some of the modules that are more complex are described in subsequent sections.

3. Validation and conflict resolution module

For any system within the scope of more than one policy; ambiguous, anomalous, or merely undesirable situations may arise. A system may enter into a state that causes *conflicting* decisions to be made by different applicable policies. A policy may be subsumed by other policies, in which case it may be redundant due to *dominance*. Policies may not have sufficient *coverage*, so that circumstances may arise in which sensors take values for which no policy is applicable, and the system does not know what to do.

3.1. Validation methods

The validation module is a library to the core policy engine. It provides a set of APIs that can be used to call the desired validation function. This is used when a new policy is added to the repository or existing policies are modified, and specifies whether the existing set of policies and the added policy together remain valid or not.

For any specific policy based system that is built using the toolkit, the system builder specifies the validation requirements in an XML file. The schema for this XML file is provided by the toolkit. The validation XML file provides for information such as the actions which are potentially conflicting in the system, the reasonable ranges of precondition parameters, and the types of tests to be performed within the system.

The following types of generic validation functions are supported in the toolkit.

3.1.1 Range check: This validation is performed on an individual policy rule to determine if the condition ranges and action attributes specified within the policy rule are allowed. For example, let a Policy be

```
if ( IPAddr= 9.2.7.0) then Action= Prioritized;
```

a range check will ensure that the IP address of 9.2.7.0 and the action of prioritized are acceptable values. The set of allowed values are specified within the validation XML file. The check is performed by having the range of permissible parameters specified as a precondition within the validation file, and the ranges of the conditions specified in the policy are checked to ensure that they lie within the desired range.

3.1.2 Consistency check: This validation verifies that the actions specified in a set of policies are conflict free. It proceeds in two steps: 1. Identify all the policy rules that could be simultaneously true; and, 2. For those policies, determine if any conflicts exist between their actions. The first step is achieved by verifying whether the hypercube formed from the conditions that are specified in the validation schema intersect. The second step is achieved by maintaining a conflict model to determine whether actions are conflicting or not. The conflict model essentially provides a lookup table to determine if two actions are conflicting. The administrator defines the relationship among actions as an input to the policy core module to build this conflict model. The conflict model can be specified in several ways in the validation files. The current version of the toolkit allows the conflict model to be specified either as an enumerated set of conflicting actions or as a set of meta-policies that specify when conflicts may occur.

3.1.3 Dominance checks: This validation finds redundant policy rules within the system. A policy is dominated if it can never be invoked because a higher priority policy rule will always be evaluated. This check is performed by implementing a subtraction method on the object class representing the precondition. Precondition A subtracted from Precondition B represents the precondition which is equivalent to B and not A. To check for dominance, we start with the assumption that the applicable precondition for any policy is its entire precondition. This is then subtracted iteratively from the higher-priority policies which have an overlapping set of preconditions. If the resulting application precondition is null, the policy is completely dominated by some combination of higher-priority policies, and can be removed from the set of policies as being redundant. The validation XML file specifies the ranges which should be used as permissible sets for each of the precondition terms.

3.2. Runtime conflict resolution

There are several different alternative approaches to run-time conflict resolution. One can employ a simple method based on priorities, if there is a natural global ranking of policies and their effects. In more complicated situations, meta-policies can be specified for conflict resolution.

4. Transformation

In general, administrators would like to deal with higher levels of abstraction in specifying policies for their systems. There is thus often a need for high-level business-oriented policies to be transformed into lower level technology-oriented policies in order for them to be used by the various components of the system. While this may sometimes require human expertise, there are situations in which a policy transformation module can be used to take the policies entered by the system administrator and convert them from one form to another before they are deployed and interpreted by the enforcement points.

An example of transformation is encountered in the support of performance or availability targets of a web-site. A policy stating the goal or objective of obtaining an uptime of 0.999 for a customer's service needs to be mapped into the number of replicated copies of a server needed for the customer. During the transformation process, the precondition or the decision part of the policy may be modified.

Transformation may be done in one of two places: either before the policies are sent to the repository, or at the decision point before the policies are sent to the enforcement point. The former type of policy transformation would typically be static based only on the policy statements themselves, while the latter could be static or dynamic, taking into account the real time state of the system.

4.1. Transformation using static rules

Static rules may be used to simplify the policy language as seen by the system administrator. In this case, an expert user, who knows the details of the system and the definitions of its various objectives, would specify how the higher level policies would be interpreted. The static-rules would specify how the precondition and cope of a higher-level policy should be modified to obtain a lower-level policy. A common form of static transformation is that in which Classes of Service (CoSs) are defined for different categories of users. A given policy would state when and for whom *gold service* would be provided. The administrators would only have to know in general terms (e.g., cost, responsiveness, throughput) the distinctions between the various classes of service. The policy transformation mechanism would then have to determine the settings of system parameters needed to obtain the particular CoS desired (bandwidth, cpu, storage, encryption, etc.).

A special case of static-rule based transformation is transformation via a table look-up. In this case, a higher-level policy of format "if precondition p then higher-layer decision b" is mapped into one or more lower-level policies of format "If precondition p' then lower-level decision b' ", while preserving the scope and priority of the same.

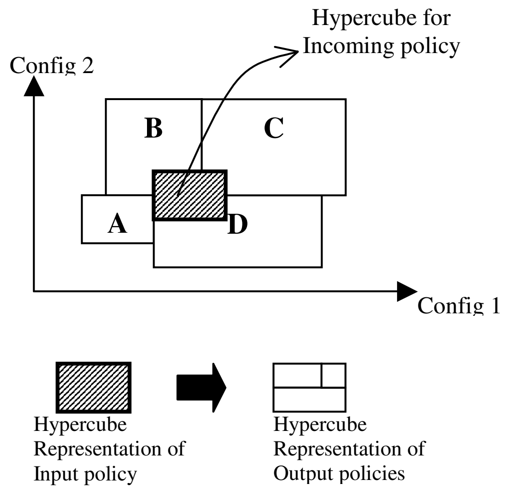
Hypercube for Incoming policy
Config 2
B
C
A
D
Config 1
Hypercube Representation of Input policy
Hypercube Representation of Output policies

Figure 3: Hypercube representation of policies

In order to obtain this transformation, the precondition term is mapped into a hyperspace whose axes are defined by the independent terms making up the precondition. A table mapping a set of regions in this hyperspace to a lower level decision forms the set of static rules for transformation. Each higher-layer policy can be mapped into one or more such connected regions into this hyper-space.

In order to ensure that a complete translation can be made, the incoming policy needs to be completely dominated (as described in Section 3.1.3) by the set of translation meta-policies. The dominance may be by a combination or two or more meta-policies. For each meta-policy which overlaps with the incoming policy, an output policy is produced where the precondition part is the overlap between the preconditions of the meta-policy and the incoming policy, and the decision part is specified by the meta-policy. When more than one meta-policy may be applicable, the meta-policy with the highest priority is used.

Figure 3 shows a simple 2-dimensional hyperspace with 4 meta-policies policies shown as A, B, C, and D. The incoming business policy being transformed is shown with a dashed pattern and is seen to overlap with meta-policies B, C and D. Three policies will be produced as the result of transformation, one each as a result of each overlapping meta-policy.

4.2. Transformation using case-based reasoning

An alternate form of transformation is to change a policy of format "if precondition p then higher-level decision d" into exactly one policy of format "if precondition p then lower-level decision b", where d is a set of measurable state of the system, and b is one or more set of configuration settings in the system. Such a transformation can be used by building a case-database which allows the mapping between configuration settings and the measurable states (or goal) within the system. The transformation module may build the case database on the basis of knowledge learned from the system behavior, or rely on a static case-database. Case based reasoning is widely used in many applications such as diagnostics, planning, prediction, and object classification [3].

When the configuration parameters needed for a new objective are required, the case database is consulted to find the closest matching case, or an interpolation is performed between the configuration parameters of a set of closest matching cases. The toolkit provides a transformation module using case-based reasoning which only requires the system developer to identify the configuration parameters and goal components of the policies being used.

The case database maintained in the toolkit contains measurements of various parameters of a system over a long period of time. Each case contains an N-dimensional set of configuration parameters and an M-dimensional set of corresponding goal values. Each case corresponds to measurements taken at one particular point in time. The cases may or may not be ordered chronologically and may or may not have associated timestamps, as selected by the system developer.

Clustering techniques are used to improve the effectiveness of case-based reasoning, and to reduce the size of the case-database. The various entries in the case-databases are aggregated into clusters, where each cluster has the same goal. The toolkit supports a couple of clustering algorithms which can be selected by the system developer. Figure 4 shows the effectiveness of simple clustering for cases consisting of two configuration parameters and one goal. The clustering process reduced the number of active cases from 21 to 3 in the example shown in the Figure.

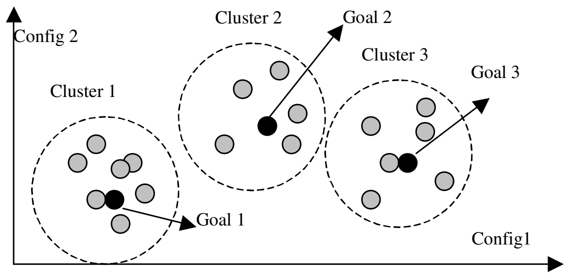

Figure 4: Clustering in a 2-dimensional space

Another way to improve the effectiveness of case-based reasoning is by reducing the number of different types of configuration parameters that need to be kept within each case. While a managed system may have a large number of configuration parameters, only a few may influence the business goal during the operation of the system. The toolkit iteratively refines the set of configuration parameters to be maintained in the case-database by assigning them a score on their correlation with the business goal, and eliminating parameters with a lower score. In a system which requires that case-history be built dynamically, the technique reduces the need for manual identification of relevant configuration parameters.

5. Design patterns

In order to ease the task of builders of policy based systems, the toolkit provides patterns for building a variety of policy enabled systems. Each pattern consists of a set of interfaces which are linked together and provide the skeletal framework for developing a policy enabled system. The toolkit provides some standard implementations of these interfaces, a subset of which need to be implemented in order to develop a customized policy based system.

Each pattern includes the definition of a set of interfaces and a main routine that ties the interfaces into a logical operation. The main routine provides the logical flow among the different interfaces. Some of the set of interfaces in the patterns are decision points, i.e. interfaces where a set of policies are consulted to make a decision. The simplest pattern included in the toolkit is the use of a policy enforcement point. Events requiring policy enforcements are sent to the policy enforcement point and the decisions of the resulting policies are returned. A couple of more complex patterns included within the toolkit are described below.

5.1. The configuration pattern

The configuration pattern can be used to build a system that lets an administrator define a set of high-level business policies, and have them transformed into a set of lower-level policies that are distributed to different devices within the network, and enforced there. The configuration pattern has been used extensively within policy based networking research for creating Quality of Service and IP-security management systems.

The configuration pattern is shown in Figure 5. A configuration pattern based on the IETF framework [4] consists of four main components: a policy management tool, a policy repository, a policy decision point, and a policy enforcement point. The system administrator enters the policies into the policy management tool. They can then be validated for correctness, and checked for potential conflicts. These policies are sent to and are stored in the policy repository as XML files. A persistent policy database is provided in the PTK Policy Core Module, but other repositories may be used (e.g., the OGSA Policy Service [5] or a network directory server accessed using the LDAP protocol [6]).

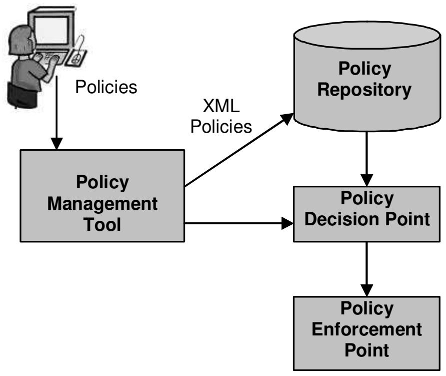

Figure 5: Configuration Pattern

The decision points (PDPs) retrieve their policies from the repository, and are responsible for interpreting the policies and communicating them to the policy enforcement points (PEPs). Depending upon the overall policy schema being employed, the PTK Decomposition Module may be needed to transform higher level policies into sets of lower level policies that can be directly used by the system. The basic decision point functionality is provided by the policy agent module within the toolkit.

The PEP is the system component that actually applies and executes the policies, and can be based on the PTK Policy Enforcement Point Module. It will evaluate its policies either periodically or on the occurrence of a specific event. The PEP and the PDP may both be located on a single device or they can be on different physical devices. Different protocols can be used for various parts of the pattern; e.g., the COPS protocol [7] or the SNMP protocol [8] can be used for communication between the PDP and the PEP.

5.2. The auditor pattern

The auditor pattern can be used to build a system that checks compliance with a specific set of criteria. Examples of this pattern would be: a policy based system that checks the configuration of a storage area network; or, one that sets data access permissions to ensure compliance with organizational privacy policies.

The auditor pattern is shown in Figure 6. It consists of four interfaces connected together. The data-scanner reads the data to be audited from the system. The data-analyzer is used to determine the set of sensors that can be passed to the decision-point. The decision-point is invoked to check policies, and the resulting decisions are used to generate events for the system.

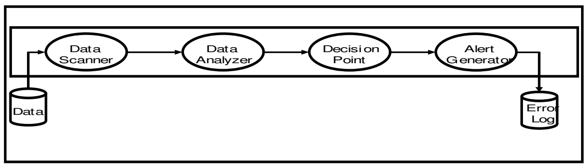

Figure 6: Auditor pattern

The auditor pattern would consist of the main class implementing the overall flow, and specific implementations of each of the interfaces. Standard implementations of the scanner include the ability to read in data from a database. The data analyzer component has an implementation that converts database entries into a set of precondition variables to be passed to the decision point. The decision point is based upon the toolkit evaluation engine, but can be replaced by any other policy evaluation system as well.

The implementation of the alert generator would consist of standard libraries that can generate an email alert, create a log-entry in a canonical format, or generate an event.

5.3. The planner pattern

The planner pattern can be used to build an application that uses policies to design or configure a new system. For example, it can be used by an on-demand or Grid environment to decide how many system resources to provide to an incoming request.

The planner pattern is shown in Figure 7. It consists of a series of interfaces, two of which are the decision points where policies are evaluated. The planner pattern has a requirements analyzer interface which takes a set of requirements and converts them into a series of preconditions. The preconditions are then used to invoke a decision point that looks at template selection policies to return a template. The template is then converted into a set of preconditions by means of a template analyzer which extracts preconditions from the template and the requirements together. A template filler decision point is used to determine any decisions regarding how templates ought to be filled in. The design creator puts the results into the template to create a design. The design writer then converts the designs into an external format, e.g. an XML format.

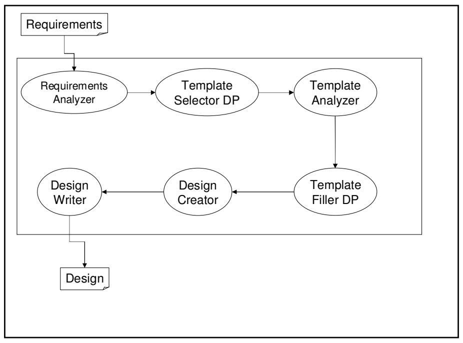

Figure 7: Planner pattern

Patterns provide a way to present toolkit users with solutions that are already pre-built to a large extent. The patterns can then be reused to build policy enabled systems rapidly. As experience with the exploitation of the toolkit is built further, we hope to define and build more generic patterns of policy exploitation.

6. Example of toolkit use

In order to better understand what functionality is needed for policy enablement, we have worked with a number of other groups on incorporating policies into their systems. One such activity involved an autonomic system to adaptively and efficiently manage resource deployment to handle unexpected workload variability [9]. This dynamic surge protection system was designed to proactively satisfy Service Level Objectives (SLO) in the face of workload surges by automatically adding the appropriate number of resources to handle a surge and then removing them when they are no longer needed.

Briefly, the dynamic surge protection system employs three technologies: adaptive short-term forecasting, on-line capacity planning, and configuration management. The forecasting approach is designed to be responsive to rapid changes, yet robust towards occasional spurious predictions (an undesirable side-effect of highly responsive predictors). On-line capacity planning determines the appropriate number of resources needed to satisfy service levels for any given workload intensity. Lastly, configuration management allows for resource adjustments, e.g., application provisioning.

The optimal setting of the parameters of operation of the dynamic surge protection system requires a detailed understanding of the controller, and hence is best suited to administrators with expert knowledge. In addition, several of the important control settings were hard-coded as static values which were a compromise over a range of operating conditions and performance expectations.

Figure 8 shows an architecture designed for integration of policy-based management into the dynamic surge protection system. The architecture consists of a policy editing tool, a policy repository, a policy agent, a policy translator, a policy decision point, and a policy enforcement point, and can be seen as an instance of the configuration pattern. The high level service objective is specified through the system administrator GUI editor and represented in a Java object that is the input to the decision logic unit of the dynamic surge protection system controller.

The components of the Policy Toolkit were used to implement the policy-based management architecture. Four rules for the controller were specified with the policy editing tool from the PTK. These rules employ high level considerations like Cost Sensitivity, Responsiveness, and Workload Variability to determine quality of service. The quality of service is expressed in terms of four classes of service: Platinum, Gold, Silver and Bronze. Each of these classes of service determines a certain level of operational performance.

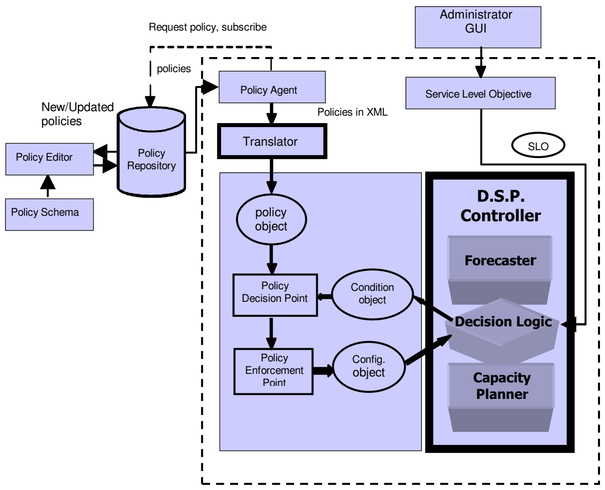

Figure 8: Policy-enabled surge protection

The detailed internal configuration parameters for the controller are fine tuned for each of the classes of service based on experience and historical data. The administrator only inputs the high level considerations affecting system behavior, such as: CostSensitivity and Responsiveness. In our implementation these take simple values of “high” or “low”. The corresponding service class is determined by the policy evaluation engine and is further transformed into the low level configuration settings for the controller by the policy enforcement point. It is important to realize that additional policies to determine class of service can be entered without bringing down the system. The specifics are “hidden” from the administrator that sets the controller objectives, and the task of the system administrator is therefore dramatically simplified.

7. Conclusions

We have presented a set of components for supporting the policy enablement of computer applications. These common functions have been developed as a Policy Toolkit that can be used across a wide variety of applications, and that simplify the task of integrating policy related methodologies into new or existing software systems.

As well as providing the necessary editing, deployment, evaluation, and management capabilities, the toolkit also includes advanced functionality like policy validation, transformation, and conflict resolution. In order to ease the task of builders of policy based systems using the toolkit, it also provides patterns for building a variety of policy enabled systems. A number of engagements have been undertaken in order to assess the usefulness of the components of the toolkit, and determining what additional capabilities might be needed.

We have found that the Policy Toolkit has made it easier and more convenient for software developers to incorporate policy-based technologies into their applications. This in turn has simplified the management and administration of these systems.

8. Acknowledgements

We would like to acknowledge the contributions of all the individuals who were involved with the Policy Toolkit and helped develop its many components. These include: James Giles, Dakshi Agrawal, Kang-Won Lee, Mandis Beigi, and David Olshefski.

9. References

[1] DMTF PCIM/e (extended Policy Core Information Model), http://www.dmtf.org/standards/documents/CIM/CIM_Schema28/CIM_Policy28-Prelim.pdf.

[2] N. Damianou, N. Dulay, E. Lupu, and M Sloman, “Ponder: A Language for Specifying Security and Management Policies for Distributed Systems”, Imperial College, UK, IEEE Policy Workshop 2002, Monterey, CA, June 2002.

[3] http://www.cbr-web.org/

[4] The IETF Policy Framework Working Group: Charter available at the URL: http://www.ietf.org/html.charters/policy-charter.html.

[5] Core Services – These are the lowest level architected OGSA service components. URL: http://www-unix.gridforum.org/mail_archive/ogsa-wg/doc00026.doc

[6] T. Howes, M. Smith, and G. Good, “Understanding and Deploying LDAP Directory Services”, MTP, ISBN 1578700701, 1999.

[7] D. Durham et. al, The COPS (Common Open Policy Service) Protocol, IETF RFC 2748, Jan 2000.

[8] W. Stallings, “SNMP, SNMPv2, SNMPv3, and RMON 1 and 2”, Addison Wesley, ISBN 0201485346, 1999.

[9] E. Lassettre, et. al, “Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions That Have Dead Times,” 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2003, Heidelberg, Germany, October 20-22, 2003.