RC23125 (W0402-138) February 26, 2004
Computer Science

IBM Research Report

Policy-Based Management for Dynamic Surge Protection

Seraphin Calo, Steven Froehlich, Maheswaran Surendra, Dinesh Verma, Xiping Wang
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Policy-Based Management for Dynamic Surge Protection

Seraphin Calo, Steven Froehlich, Maheswaran Surendra, Dinesh Verma and Xiping Wang
IBM T. J. Watson Research Center
P.O.Box 704, Yorktown Heights, NY 10598
scalo, stevefro, suren, dverma and xiping@us.ibm.com

Abstract

Policies are increasingly being used to manage complex systems. This paper presents our work on policy enablement of a dynamic surge protection system, which predicts the traffic changes in an IT environment and then balances an application's resources based on a set of pre-defined policies. We aggregate the controller settings of the system into classes of service that can be more intuitively determined through administrative policies. The detailed settings of the elements of each class are defined and maintained by the use of a policy management tool. The controller automatically allocates or de-allocates application server resources in order to satisfy time varying workloads based on the input service level objectives. This not only externalizes the controller parameter settings but also allows the modification of the policies to change the behavior and strategy of the dynamic surge protection system without recoding the controller. As a result, the administrator's tasks can be dramatically simplified, as the system adapts to changing environments.

1. Introduction

Today's IT systems have widely varying demands for resources due to unexpected surges in subscriber accesses. New applications are deployed, but their resource demands are unknown. The typical approach for dealing with these problems is to over-provision and/or manually re-allocate resources. Unfortunately, both approaches are undesirable due to cost factors (equipment, licenses, etc) and the need for expert operators. In addition, resource actions often involve lead times, such as server warm-up or cool-down periods, resulting in delays between action initiation and effect.

As part of IBM's thrust in autonomic computing [1][2][3], a previous work introduces a system to adaptively and efficiently manage resource deployment to handle unexpected workload variability [4]. This dynamic surge protection system is 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 (described in more detail in [4]) 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 are hard-coded as static values which are a compromise over a range of operating conditions and performance expectations.

Policy-based technologies are increasingly being used in the management of networks and distributed systems [5]. Policies are rules governing the choices in the behavior of a system. Explicitly separating policies from the system components that interpret them allows the modification of the policies to change the behavior and strategy of the management system without changing the management software itself. The management components can then adapt to changing requirements by disabling policies or updating policies without shutting down the system. This is obviously very desirable for the dynamic surge protection system, since it is meant to provide continuous adjustment of resources to meet service level

objectives. The policy-enablement of its operation avoids potential downtime when external conditions change, and significantly simplifies its management. To this end, we have therefore extended the system controller to accept pre-defined policies, and have incorporated policy management tools into the overall system for the editing, storage, and deployment of policy information.

This paper describes our work on the policy enablement of the dynamic surge protection system, including a policy framework and its implementation. The paper is organized as follows: Section 2 presents a description of the dynamic surge protection system; Section 3 describes the extensions to that system for policy enablement; Section 4 presents experimental results obtained on our research testbed; and, Section 5 presents conclusions and future work.

2. Dynamic surge protection

Since the dynamic surge protection system is presented in detail elsewhere [4], we provide a brief description of the research testbed which also highlights elements pertinent to policy enablement.

The system is organized into three layers as shown in Figure 1. The Application layer provides the business function. In this work, a two-tier web application with one or more application servers (IBM's Websphere Application Server 5.0) and a database server (IBM's DB2 v8.1 database) is used. The application deployed on this testbed simulates the supply chain management of a manufacturing company.

A key feature of the system is that the application tier can scale horizontally without requiring system shutdown. The testbed also includes a workload driver, controller and deployment manager on another machine. Both Controller and Deployment Manager code are lightweight and incur minimal CPU load.

The Deployment Manager interfaces with the application layer, and has the appropriate hooks for monitoring and configuring the application layer. Transaction rates (business operations per sec) and response times are monitored, while configuration management (provisioner) is focused on the addition or removal of application servers.

The Controller monitors the application layer state and initiates appropriate actions if an SLO (e.g., response time) violation is anticipated or if the SLO can be satisfied in a more cost-effective way.

Figure 2 shows the control and data flow for dynamic surge protection. Workload data from monitoring are input to the forecaster, which predicts future workload. The capacity planner uses the predicted workload and SLO to determine the appropriate number of application servers needed. The

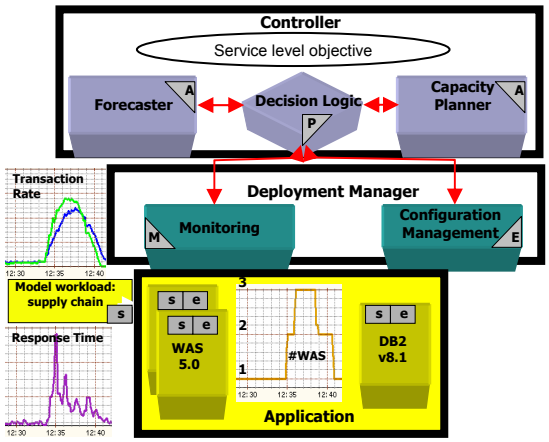


Figure 1. Architecture for dynamic surge protection.

decision logic manages the information flow and determines the resource adjustments (by comparison to the deployment state data). These adjustments are effected by the provisioner.

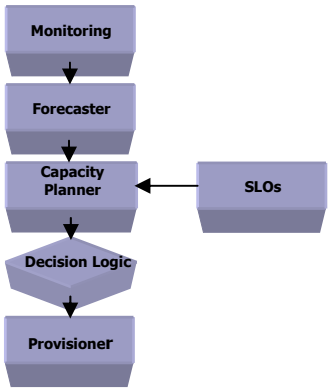


Figure 2. Control and data flow.

Key technologies employed by the dynamic surge protection system are adaptive forecasting, on-line capacity planning and rapid configuration management. The adaptive forecaster employed is based on a non-seasonal autoregressive predictor with variable order (for robustness) and uses very short history (hence it is able to learn quickly). The on-line capacity planner is queuing-model based and is not computationally intensive. Rapid application server

provisioning leverages WAS 5.0's cellular cluster capability and uses the WAS startServer/stopServer commands.

As reasoned in the previous work [4], a prediction horizon H which is approximately equal to P + S, where P is the control interval, and S is the resource addition lead time, is expected to be sufficient to proactively avoid SLO violations. Here, as before, P = 10 sec, S ~ 40 sec, and H = 60 sec. Implicit in this reasoning is that once an application server is added, it can take load as effectively as the other active servers. However, in practice this is not always the case, as there is some period (about 20 – 30 sec) where it services the load quite poorly (due to class loading, etc). Hence, response times can be temporarily worse, especially with the simple round-robin scheduling option used with the application server's workload manager. Another observation is that the stopping time for an application server can sometimes be quite long (3 min – which corresponds to a preconfigured timeout).

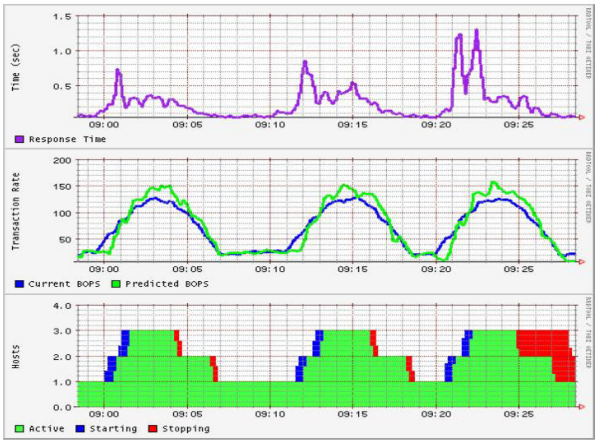


Figure 3. Typical performance from the dynamic surge protection testbed.

In Figure 3, typical testbed performance is shown. The top panel shows the response time, while the middle panel shows the actual and 60 sec ahead predicted transaction rates (business ops per sec) and the bottom panel shows the application servers in their different states as they go from idle to active (i.e. starting) and the converse (i.e. stopping). The key features in Figure 3 are that once a transaction rate surge starts, the predicted rate, while lagging at first, quickly adapts and provides useful leading information about the surge. Additional application servers are made active to handle the additional work, hence keeping the response time low (mostly < 1 sec), and are removed when the surge subsides. Note that these surges occur randomly, and the short-term forecaster does not retain any "memory" of the preceding surge.

3. Policy enablement for dynamic surge protection

The core of the dynamic surge protection system is the controller. The goal of the controller is to maintain service level objectives in the face of time varying workload. In order to achieve this goal, a set of internal rules/parameters is used to govern controller decisions. These rules/parameters determine how to dampen controller actions, how aggressively to respond to a workload upswing or how to handle insufficient guidance from the forecaster and capacity planner.

These rules are at a low level in terms of their specificity to the controller software, and programmed into the control logic in the first prototype [4]. As such, they can only be modified by expert programmers, since it requires a deep understanding of the system control logic to make any modifications. Therefore, coding the controller settings into the control logic results in an inflexible system design and makes it very difficult to administer the system. It is possible to externalize controller settings through a configuration file, but this is not the best solution as the administrator would still need to understand the controller in detail.

The more appealing alternative is for the administrator of the dynamic surge protection system to use a policy-based management tool that allows configuration of dynamic surge protection at a higher level of abstraction, instead of having to set individual internal low-level configuration. This way, it not only simplifies the system administration but also ensures that the system always performs at its best by using the expert fine-tuned configuration parameters guided by a set of policies. More importantly, the use of policies makes it possible to dynamically change the behavior of the dynamic surge protection system without changing its code. These characteristics reduce system complexity, while permitting an efficient control of service level objectives. Furthermore, policy can be dynamically modified to be adaptive to system requirements.

To abstract away the complexity of internal configuration parameters, we aggregate internal settings of dynamic surge protection into classes of service, so that the controller configuration can be determined more intuitively through pre-specified administration policy decisions. For example, the system administrator only needs to decide how cost

sensitive the controller should be or how responsive the controller should be, while the detailed individual internal configuration parameters are obtained through policy transformations that map the higher level service objectives to lower level configuration settings.

Figure 4 shows a joint 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. 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. A detailed description of each component is given below.

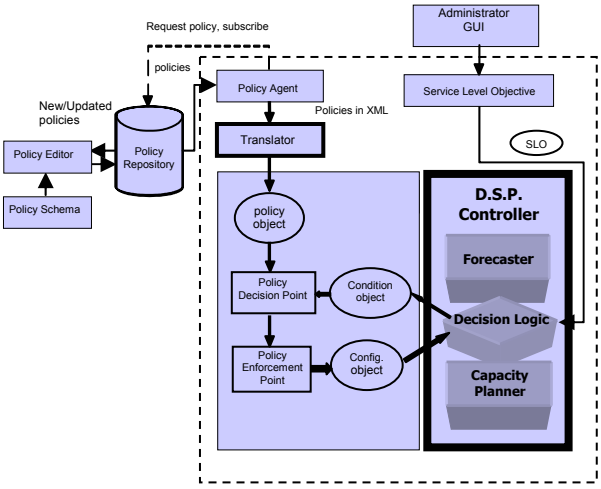


Figure 4. Policy-enabled dynamic surge protection.

3.1. Policy specification

Policies are defined as XML documents associated with a policy schema file that specifies how to validate the semantic correctness of the policy document. A "4-tuple" policy specification is used, consisting of components defining the Scope, Pre-condition(s), Priority and Action(s) associated with the policy. The Scope of a policy is an indication of the type of resource manager that is intended to be influenced by the policy. The Pre-condition component defines the situations under which a particular policy is to apply. The priority of a policy indicates its relative importance with respect to other policies with the same scope, and can be used by the policy decision point for conflict resolution in the event that multiple policies are satisfied simultaneously. The Action component describes specifically what is to be done when the policy is applied to the system.

An example policy for the controller using the "4-tuple" representation can be described as follows:

Scope:
    Controller
Precondition:
    CostSensitivity == high &&
    Responsiveness == low
Action:
    ClassOfService = Bronze
Priority:
    Value = high

In our implementation, we use a policy editing tool to create, edit and view policies. The editing tool also validates policies at policy creation time and provides a way to control policies when they are ready to be deployed.

3.2. Policy repository

The policy repository stores policies either locally or remotely. In addition, it also performs policy validation, static conflict resolution and policy transformation, and distributes policies over a policy-based management system. It is designed to interact with the policy editing tool and the policy agent.

3.3. Policy service agent

The policy agent is responsible for retrieving policies requested by the policy decision point from a policy repository. It first subscribes to services being provided by the policy repository, and then obtains policies of interest. When a policy is modified, the policy agent will notify the users of that policy about changes through a notification mechanism.

3.4. Policy translator

The policy translator parses an XML-based policy document, and converts it into a Java policy object. The resulting Java policy object is then used for all the policy processing functions, such as validation, evaluation, etc.

3.5. Policy decision point

The policy decision point evaluates policies and provides decisions to the controller of the dynamic surge protection system in order to affect its behavior. As shown in Figure 4, the Pre-conditions are obtained from the decision logic unit of the controller. Whenever the Pre-conditions evaluate to true for any

of the pre-defined policies associated with the controller, the actions indicated in that policy will be applied to the controller. In the dynamic surge protection system, that means that the internal parameters of the controller will be set to those defined in the class of service determined by the policy. In the event that multiple policies are appropriate at the same time, any potential conflict or inconsistency must be resolved. This can be done based on the priority value of each policy or using a meta- policy based approach [7].

3.6. Policy enforcement point

The policy enforcement point is the component of the system that enforces the selected policy by passing a new Configuration object to the decision logic that contains the low level configuration settings that are understood by the controller of the dynamic surge protection system.

4. Experiments

In support of the IBM autonomic computing strategy, the Policy Technologies Group at the IBM T. J. Watson Research Center, has developed a set of software development tools called the Policy ToolKit [6]. Figure 5 shows the software architecture of the Policy ToolKit. It consists of a set of modules performing specific functions plus a set of common use classes. The policy editor module, which generates a customized policy editing GUI for different applications, can be used to specify policies for a given policy-based application. The validation module can be used to perform a set of validation checks on a group of policies. The decomposition module translates a high level policy into a low level resource configuration. Conversely, the composition module translates a low-level resource configuration into a high-level policy. The policy agent module can be used to interface with a policy repository. The policy enforcement module can be used to create a policy enforcement point which executes policies satisfying a set of specified pre-conditions. The policy conflict resolution module can be used to detect and resolve policy conflicts among a group of policies. The policy rule hierarchy module can be used to merge policy groups within the same policy rule hierarchy. The policy core classes represent policy rules, pre-conditions, actions, etc.

In addition, the Policy ToolKit contains a policy utility module consisting of a set of Java helper classes to ease policy processing. We note that XML and XML schemas play an important role in the Policy Toolkit. The XML file is used as a mechanism to define policies, and to import and export policies into and out of the Policy Toolkit environment, while the XML schema file is used to define the structure of the policy files.

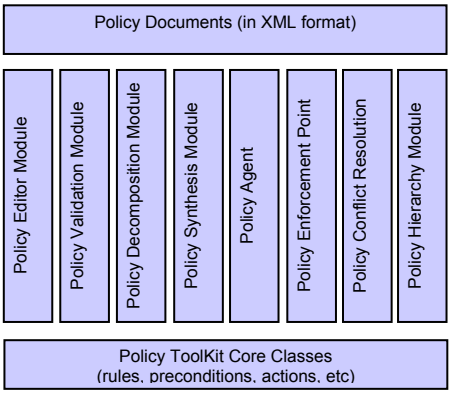


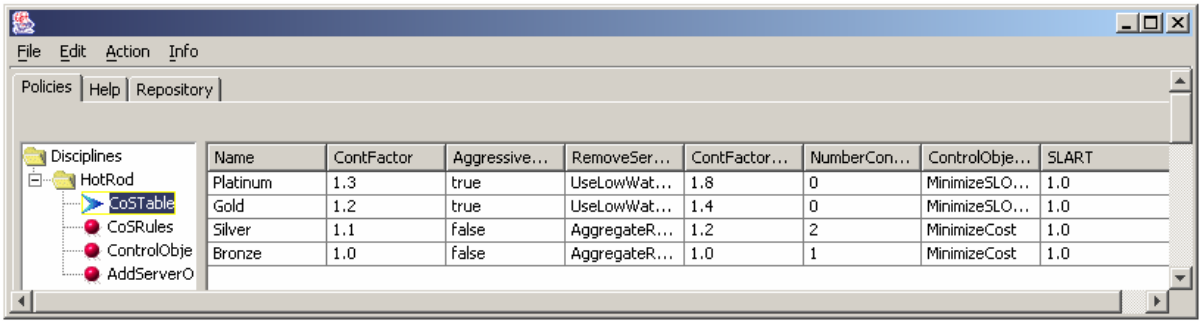
Figure 5. Software architecture of the policy toolkit.

The Policy ToolKit is written in pure Java and supports all the functions described above. The ToolKit can be used across a wide variety of applications and can simplify the task of developing or integrating policy related methodologies into new or existing software systems.

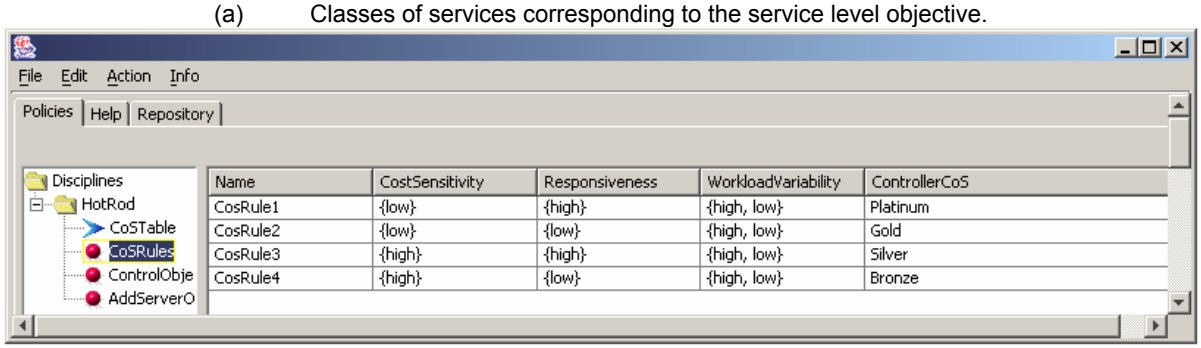
Using the Policy ToolKit we have designed and implemented policy-based management for the controller of the dynamic surge protection system. Figure 6 (b) shows a graphical view produced by the policy editing tool displaying four rules specified for the controller. These rules employ high level considerations like Cost Sensitivity, Responsiveness, and Workload Variability (not used in the current prototype) to determine quality of service. Four classes of service were defined for the controller of the dynamic surge protection system. These are: Platinum, Gold, Silver and Bronze as shown in Figure 6 (a). Each of these classes of service determines a certain level of operational performance.

The detailed internal configuration parameters for the controller are fine tuned for each of the classes of service based on experience and historical data. We note that one of the attributes in the service class definition (ControlObjective) is not used in the work described here, but remains as a place holder for future work. Most of the other attributes (e.g., ContFactor) are there to provide robustness to poor forecasting/capacity planning. The administrator only

inputs the high level considerations affecting system behavior, such as:



(a) Classes of services corresponding to the service level objective.



(b) Four rules defined for the controller.

Figure 6. View of the policy editing tool.

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 (e.g., what class to use if both CostSensitivity and Responsiveness are both set to "medium") 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.

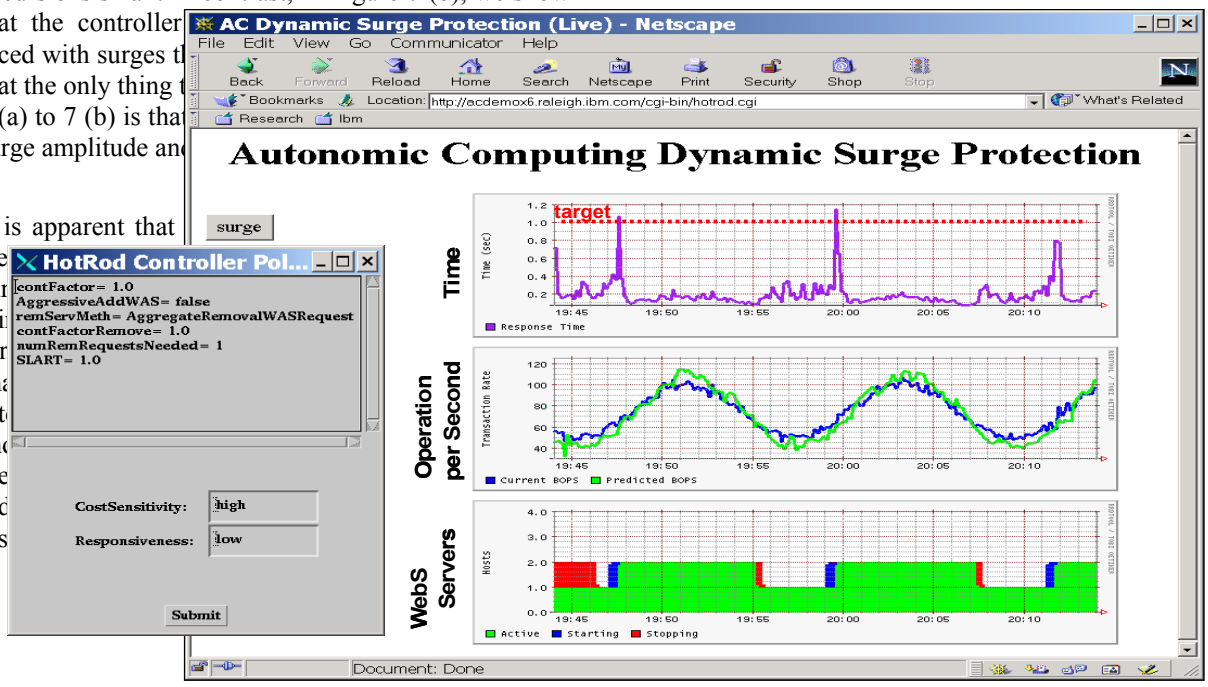
The simple policy administration panel used with the prototype is included in Figure 7 (a), (b), and (c). The bottom part displays the high level input parameters from the system administrator, while the upper part depicts the corresponding low level configuration settings produced by the policy enforcement point as shown in Figure 4.

Figure 7 also shows the results of the experiments conducted on our research testbed. The middle graph of Figure 7 (a), (b) and (c) plots the actual and predicted business operations per second (BOPS), the metric used to characterize workload. The bottom graph of Figures 7 (a), (b) and (c) shows the changes of state and the number of application servers allocated in response to the actual and forecasted demands on the system. The top graph of Figures 7 (a), (b) and (c) depicts the effect that these actions have on response times.

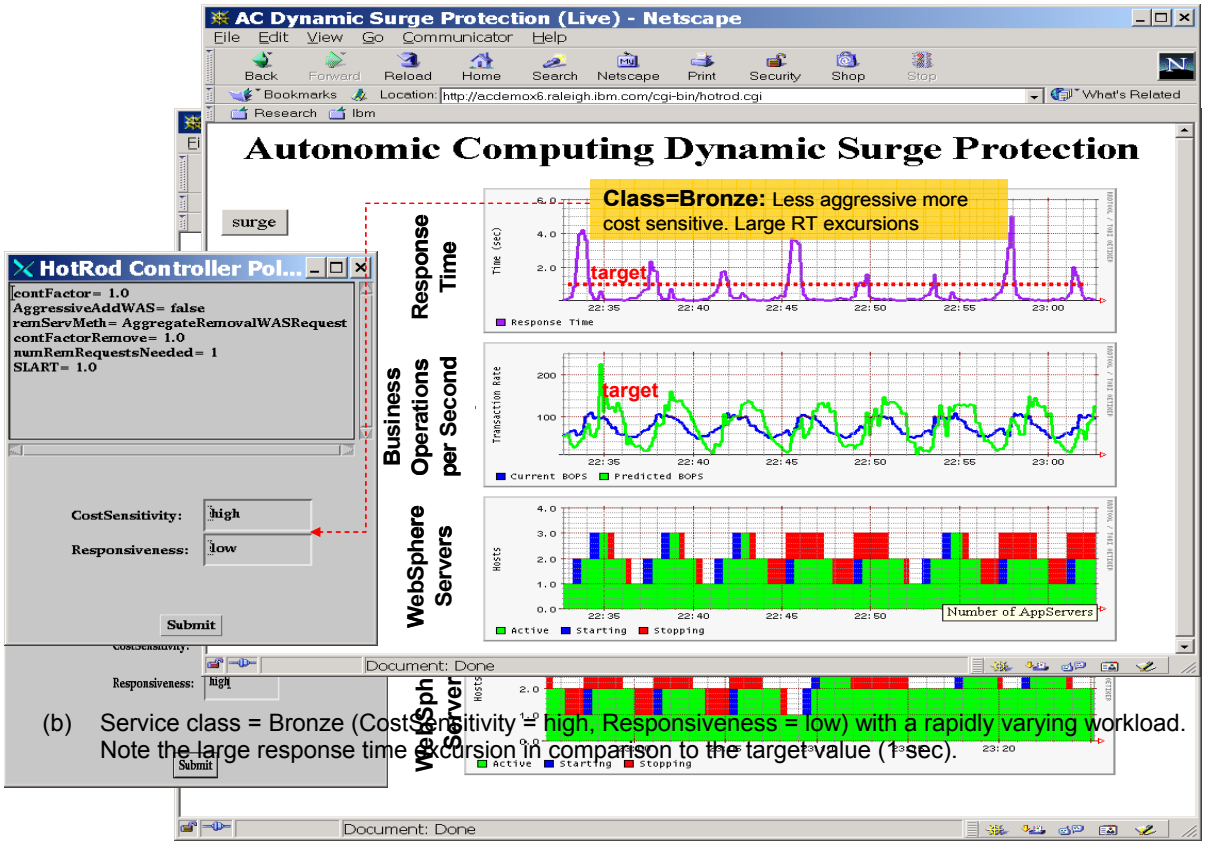
To effectively demonstrate the value of policy enablement within the time span of the controller display (30 min), we actually use a workload with periodic surges, although the system is designed to deal with random surges. In Figure 7 (a), we show results using a low class of service (Bronze), which corresponds to administrator preferences for high CostSensitivity and low Responsiveness. This is the most economical controller setting, and is appropriate for workload surges with relatively low ramp rates. The predicted transaction rate tracks the actual transaction rate quite well, and the additional resources are brought on-line in time to keep response time

excursions small. In contrast, in Figure 7 (b), we show
that the controller
faced with surges t
that the only thing
7 (a) to 7 (b) is that
surge amplitude an

It is apparent that
we
sur
mi
for
cha
int
an
the
(ac
res



(a) Service class = Bronze (CostSensitivity = high, Responsiveness = low) with a slowly varying workload. The system is performing as expected, and the response times as behaving well with regards to their target value (1 sec).



(b) Service class = Bronze (CostSensitivity = high, Responsiveness = low) with a rapidly varying workload. Note the large response time excursion in comparison to the target value (1 sec).

(c) Service class changed from bronze to platinum (CostSensitivity = low, Responsiveness = high) with rapidly varying workload. Note that response time excursions are greatly reduced with the platinum service class. The added performance does come at the cost of deploying a higher average number of WAS servers.

Figure 7. Performance of the dynamic surge protection system.

In Figure 7 (c), the administrator preferences are changed to low CostSensitivity and high Responsiveness. This results in a class of service that is more responsive to more rapidly varying workloads, and is also more apt to use additional resources to minimize temporary response time excursions. With the change to this higher service class (Platinum), we note that again the response time is quite well behaved, but this additional performance comes at the cost of higher average server usage (about 2.5 in Figure 7 (c), vs. about 1.7 in Figure 7 (a)). As indicated in Section 3, some controller rules/parameters are used to accommodate situations where guidance from the forecaster/capacity planner is poor. A typical approach to uncertainty is to use contingency factors in determining when to add/remove servers. Additional "safety" is provided by using a lower threshold (higher contingency factor) when removing a server in comparison to adding a server. This feature is enabled by setting RemoveServerMethod = UseLowWaterMarkContFactor (Gold and Platinum service classes). The downside of this feature is that it tends to keep more servers in the active state than a situation where servers are added/removed based on the same threshold (Silver and Bronze service classes).

This prototype was shown to several different audiences in IBM and it is interesting to note their reactions/feedback. The audiences themselves were varied, ranging from policy experts to developers, system administrators, and executives. Most of the discussions revolved around what type of inputs (e.g., here CostSensitivity and Responsiveness), if any, should be exposed to the administrator. The actual service classes (Platinum to Bronze) did not generate as much discussion, which is probably not surprising, since they have to do with actual controller settings for performance.

One opinion was that the service class should be directly chosen by the administrator, without having the higher level considerations. A motivation behind this is that CostSensitivity and Responsiveness cannot really be considered to be orthogonal inputs – i.e. a highly responsive system would typically be less cost sensitive.

Another opinion was that these considerations should not serve as inputs, but should be derived from a higher level policy or service level agreement (SLA). For instance, an SLA stating that 80% of the response time (RT) measurements taken in a defined time window must be less than RT_{SLA}, could possibly map onto the Bronze service class. Alternatively, if the requirement was that 95% of these response time measurements be less than RT_{SLA}, the mapping would be to the Platinum service class. This would be ideal in a sense that a business policy would be automatically decomposed into IT policies. The challenge is that the appropriate decomposition is not easy to derive, especially since it needs to somehow capture the workload variability (the capacity planning tool we use is not designed for transients). One approach to this would be to do online model building and learning to elucidate the appropriate service class corresponding to the SLA parameters. Note that the usefulness of learning is not just limited to determine service classes, but also the values of the low level configuration parameters. For instance, by measuring forecast error together with associated RT over a period of time, it should be possible to adjust the ContFactor to accommodate high forecast error.

Other points of discussion centered around the use of unsolicited decisions. This is more involved than simply using some other input in the policy pre-conditions (e.g., variables not entered by the administrator, such as time of day), since the current implementation of the architecture we show in Figure 4 is geared towards solicited decisions, i.e., the controller asks for the configuration object at the beginning of every 10s control interval. In the unsolicited case, the controller would continue to use the same configuration object until it was explicitly updated by some other management component. This would not make a significant difference in the present system, since these objects are small; but, this might be important in environments where large amounts of data are involved in the decision process.

We note that this is a separate consideration than that of solicited versus unsolicited policies. In the current architecture, the configuration object is updated whenever a new condition object is received. It is also updated whenever a change is made to the policy object. The policy subscription mechanism thus supports unsolicited updates.

5. Conclusions

This paper has presented a policy-based management scenario, which shows how the controller of the dynamic surge protection system was policy-enabled, allowing it to dynamically allocate application server resources based on a set of pre-defined policies in order to achieve its required service level objectives. The internal settings of the controller were aggregated into classes of service that could be more intuitively determined by administrator policies. Since the detailed settings of the controller were abstracted away

by policies, the task of the system administrator has been dramatically simplified. The integration of policy-based management into the controller of the dynamic surge protection system also separates its control settings from its control logic, making it possible for the controller to adjust itself to dynamically meet a range of operating conditions and also performance expectations by simply modifying its policies.

A policy management tool was developed based on a set of common tools provided by the Policy ToolKit. We have also conducted a number of experiments on a testbed system to gain insight into the characteristics of dynamic surge protection and the effects of policy-based management. Overall, we have found it to be well behaved, although the policies designed for the controller of the dynamic surge system were relatively simple.

Our future work will extend the current system to include more complicated policies for the handling of multiple workloads, incorporating tuning as well as provisioning actions. Conflict detection and resolution will also be addressed in order to resolve potential inconsistencies caused by interacting sets of more complex policies.

6. Acknowledgements

We would like to acknowledge the contributions of all the individuals who were involved with the Dynamic Surge Protection System and the Policy Toolkit. Mandis Beigi was particularly helpful in developing policy schemas for different versions of the prototype. We would also like to recognize Nagui Halim who headed the Distributed Computing Department, and Richard Telford, Director of Autonomic Computing, who both encouraged the policy-enablement effort.

7. References

[1] IBM Autonomic computing, Creating self-managing computing systems (http://www.ibm.com/autonomic), 2003.

[2] Autonomic Computing Roadmap V1.0, September 26, 2002.

[3] An Architectural Framework for Autonomic Computing v2.0, October 22, 2002.

[4] 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, Proceedings.

[5] Morris Sloman, Jorge Lobo and Emil Lupu, "Policies for Distributed Systems and Networks," International Workshop, POLICY 2001 Bristol, UK, January 29-31, 2001, Proceedings.

[6] IBM Policy ToolKit High Level Design V1.0, July 2003.

[7] E. Lupu and M Sloman, "Conflicts in Policy-Based Distributed Systems Management," IEEE Transactions on Software Engineering, Vol. 25, No. 6, Nov/Dec 1999.