

# A Self-healing Web Server Using Differentiated Services

Henri Naccache<sup>1</sup>, Gerald C. Gannod<sup>2,\*,\*\*</sup>, and Kevin A. Gary<sup>3</sup>

<sup>1</sup> Dept. of Computer Science & Engineering, Arizona State University  
Box 878809, Tempe, AZ 85287

henri@asu.edu

<sup>2</sup> Dept. of Computer Science & Systems Analysis  
Miami University, Oxford OH 45056

gannodg@muohio.edu

<sup>3</sup> Division of Computing Studies, Arizona State University  
7001 E. Williams Field Rd., Mesa, AZ 85212

kgary@asu.edu

**Abstract.** Web-based portals are a convenient and effective mechanism for integrating information from a wide variety of sources, including Web services. However, since availability and performance of Web services cannot be guaranteed, availability of information and overall performance of a portal can vary. In this paper, we describe a framework for developing an autonomic self-healing portal system that relies on the notion of differentiated services (i.e., services that provide common behavior with variable quality of service) in order to survive unexpected traffic loads and slowdowns in underlying Web services. We also present a theoretical performance model that predicts the impact of the framework on existing systems. We demonstrate the framework with an example and provide an evaluation of the technique.

## 1 Introduction

In this paper we present a preliminary investigation into a self-healing framework implemented within a Java-based Web portal. As the load on the portal increases, and the response times surpass those set forth in the service level agreement (SLA), the framework directs the underlying adaptive-content aware components to lower their output resolution. The lowering of the output resolution of the components minimizes the service demands of the components on the system and allows for higher request loads to be handled within the same SLA.

Along with the preliminary implementation, we present a Queuing Network (QN) analytical model that shows the expected increase in request load handling that can be gained by implementing the self-healing framework. The sample

---

\* This author supported by National Science Foundation CAREER grant No. CCR-0133956.

\*\* Contact Author.

portal implementation relies upon the database-driven nature of modern web applications.

The benefits of our framework are tangible when the site is hit with a higher than expected request load, usually the result of a *flash crowd* (the “Slashdot effect”) [1] that would normally slow or stop a site from responding to requests. With our self-healing approach, the server can handle a much higher maximum load as it lowers the resolution of the web pages to just the minimum needed to convey the requested information. This approach requires a small investment in software that can reduce the cost of hardware needed to run a portal website. Normal capacity planning requires that the server support the peak load rather than the average load [2]. With our self-healing framework in place, the hardware requirements can be defined to meet the SLA of the average request load at full resolution and the peak load at the lowest resolution.

While admission control and queue management have been shown to improve throughput [3] and response times [4], the cost incurred is request refusal. We consider request refusal to be the worst experience an end-user can have. Most visitors in a flash crowd will be looking for the same information. In order for the site to survive the flash crowd it must allocate its resources optimally and only return the lowest feasible resolution of information while not refusing any valid requests.

The remainder of this paper is organized as follows. Section 2 describes background material on autonomic computing, adaptive content and QN models. The self-healing portal framework is presented in Section 3. An evaluation of that implementation is presented in Section 4. Section 5 discusses related work. Finally, Section 6 draws conclusions and suggests future investigations.

## 2 Background

This section describes background material in the areas of autonomic computing, adaptive content and QN models.

### 2.1 Self-healing Systems

The term *autonomic* comes from the autonomic nervous system found in mammals and other higher order creatures [5]. This aspect of the nervous system allows for necessary body functions to perform without conscious thought given to them. The fundamentals of autonomic computing revolve around self-managing components. In this context, inter-component collaboration is defined in a self-managing manner. The goals of autonomic computing are to minimize human intervention in system administration and to maximize reliability and system uptime. In order to do so, IBM has defined four fundamental features of components: *self-configuring*, *self-healing*, *self-optimizing* and *self-protecting* [6]. Self-healing systems “discover, diagnose, and react to disruptions”. Self-healing systems must be able to recover from the failure of underlying components and services. The system must be able to detect and isolate the failed component, fix

or replace the component, and finally reintroduce the repaired or replaced component without any apparent application disruption. Self-healing systems need to monitor components in order to predict problems and take action before complete component failure takes place. The objective of self-healing components is to minimize the number and duration of outages in order to maintain high levels of application availability.

An autonomic component consists of two integrated parts: a managed element, the underlying service or component, and an autonomic manager. [7] The manager is in charge of maintaining status information about the managed element and making sure that the managed element remains healthy.

Our research into differentiated services at the web application level is an example of a self-healing autonomic system.

## 2.2 Adaptive Content

An approach to QoS management at the web server level is to adapt the content in order to minimize bandwidth usage. The common approach is to re-encode multimedia files into lower-quality, and therefore smaller, files. This has been proposed by Bellavista [8] among others. The majority of this research was done in the early days of the internet before the current prevalence of dynamic database driven websites and the performance issues that come along with them.

Abdelzaher [9] offered an approach to minimize bandwidth requirements by providing two completely distinct websites on the same server. The sites offer the same base content, but each has a different resolution of content. This differs from the multimedia-only encoding in that it can also offer differentiated text-based content. In this system the two sites have to have similar file structures and file names, and have to be manually created by the webmaster. Depending on the current QoS level the server responds to a request with a file from either “tree”.

We apply adaptive content techniques to dynamic multi-tiered web applications in order to maximize throughput at the server level.

## 2.3 QN Models

Analytic performance models are used to predict the response of a system to various configuration and design changes. They are composed of a set of computational algorithms that use actual workload parameters to compute the performance characteristics of a system. Using various analytic models, one can identify bottlenecks and estimate upper bounds on response times.

Queuing Network (QN) models are one way of creating an analytic performance model of a system. Menasce [10] defines a QN model as a collection of interconnected queues. Queues include both the resource providing the service and the waiting line to access that resource. The QN is used to model a system and estimate the performance impacts of design decisions. Two parameter types are used in creating QN models: workload intensity and service demands. Workload intensity provides an indication of the load of the system, and service demands are the average response times of specific resources in the system.

QN models can be used to represent multiclass systems. A multiclass system is one that supports more than one type of request. Services that output multiple resolutions can be modeled as multiclass systems. Open and closed networks represent two types of request arrival distributions. Open networks assume that request arrivals are uniformly distributed and throughput is used as a parameter to the model, while closed networks are used to model bulk or batch jobs where there is an assumption of a near constant number of requests in the system.

Response times  $R_c$  (Eq. 1 [10]) are calculated in a multiclass open QN based upon the utilization of each device ( $U_i$ ) and the service demand of the user class on the device  $i$ :  $D_{i,c}$ .

$$R_c = \sum_{i=1}^K \frac{D_{i,c}}{1 - U_i} \quad (1)$$

Where the units for  $R_c$  are seconds,  $D_{i,c}$  are seconds/request,  $c$  is the user class,  $i$  is the device and  $K$  is the total number of devices in the system. The total utilization of a device  $U_i$  is the sum of the utilizations due to all classes (Eq. 2). When  $U_i = 1$  the device is fully utilized. The per class utilization is a product of arrival rates  $\lambda_c$  and service demands.

$$U_i = \sum_{c=1}^C U_{i,c} = \sum_{c=1}^C \lambda_c * D_{i,c} \quad (2)$$

In this paper we use a QN model with a modified response time formula to estimate the impact our framework will have on existing web applications.

### 3 Approach

This section describes the proposed framework for developing QN models with resolution factors and self-healing portal systems.

#### 3.1 QN Model with Resolution Factors

In a QN model each device that impacts the performance of the system is modeled. Each device must have the service demands ( $D_{i,c}$ ) measured for each class of request, where  $c$  is the user class, and  $i$  is the device. The service demands are represented as the time spent for the device to complete the request. The goal of the adaptive content framework is to lower the service demand on the most bottleneck-prone parts of the system. By reducing the resolution, the service demand on a device will go down by some factor.

We represent the impact of lowering the resolution as the resolution factor  $F_{i,x}$ , where  $x$  is the resolution level. For a full resolution request  $F_{i,x} = 1$ . The resolution factor is applied to the service demand in order to predict the response time  $R'_c$  for a given user class, as shown in Eq. (3).

$$R'_c = \sum_{i=1}^K \frac{D_{i,c} * F_{i,x}}{1 - U'_i} \quad (3)$$

Where  $c$  is the user class,  $i$  is the device and  $K$  is the total number of devices in the system. The total utilization of a device  $U'_i$  is the sum of the utilizations from all classes. The per class utilization is a product of arrival rates and service demands.

$$U'_i = \sum_{c=1}^C U'_{i,c} = \sum_{c=1}^C \lambda_c * D_{i,c} * F_{i,x} \quad (4)$$

Where  $\lambda$  is the arrival rate of requests of the class  $c$ . The first summation in Eq. (4) states that the utilization of a device is the sum of utilizations from all classes in the system. The second summation, which represents the per class utilization for a device, is the product of the arrival rates, the service demand and the resolution factor.

Our goal is to create a set of resolution factors that we can use to predict the impact of the self-healing framework on an existing web application. We will accomplish this by analyzing existing web applications, implementing our framework and measuring the impact on the service demands.

### 3.2 Portal System

We have developed a framework for developing self-healing portal systems based on monitoring service latency and overall system rendering performance. In contrast to the approach by Menasce et al. where QoS is managed by individual services [11], our approach places the burden of achieving a negotiated level of service on a portal application. As such, applications built within this framework provide user-level guarantees of QoS rather than component level guarantees, all of which can be managed dynamically at run-time.

Figure 1 shows the conceptual architecture of the approach that we have developed for creating self-healing portal systems that manage and monitor the use of several services. In the figure, the portal server is shown in the middle part of the diagram, clients on the left, and services, running on separate machines, shown on the right. As indicated in the diagram, a portal server in our framework is made up of a portal hosting system, autonomic monitor, and several self-healing portlet wrappers (one per service rendering portlet). The approach works as follows. The autonomic monitor continuously monitors the portal system state to determine whether the portal system is operating within certain specified parameters. The frequency of the checks typically corresponds with the frequency of client requests, but it may vary. When a request is made upon the portal, a QoS portlet wrapper checks the load on the system using a query to the monitor. Based on the load, the recent response times of the service, and the SLA for the user, the portlet wrapper potentially modifies the request and makes call to the corresponding service.

We have developed our approach with the following concerns in mind. First, we view services as potentially uncontrollable assets. The ability or inability to deliver specific levels of service is subject to network latency and load of the server or servers providing the service.

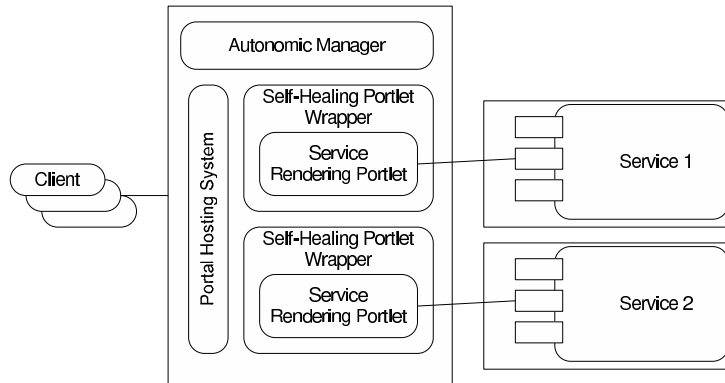


Fig. 1. Portal Conceptual Architecture

Second, for the end-server (i.e., the portal server providing some application), the potential variability in level of service caused by the aforementioned factors can be outweighed by the time to render the local application. That is, we take a global approach to providing an improved response time for an application by measuring time to render an application that utilizes services rather than optimizing individual services. As such, our approach for providing QoS-aware portal systems is based on the following concepts.

**Monitoring:** In order for an autonomic portal to properly meet service level agreements, it must be capable of monitoring its current load. Based on system upgrade and downgrade policies, this information is used to determine how information is requested from channels or service providers, and how that data is subsequently rendered in the portal.

**Feedback:** As stated by Bouch et al. [12], user experiences and acceptance of variations in performance are affected by feedback. In our approach, we use visual feedback to provide users with an indication of system state in order to provide explanations of why behaviors of services may differ over time.

**Differentiation:** We consider two forms of differentiation within our approach. First, at the user level, there are different user classes that each have different SLAs. The QoS levels that each user class receives are selected according the SLAs. Second, at the service level, we consider different service resolutions. The different resolutions are intended to affect the overall size of a package returned by a service as well as the amount of processing required to render a portal page.

## 4 Evaluation

In this section we describe the process by which we evaluated the self-healing portal system. Figure 2 provides an overview of the deployment of the portal

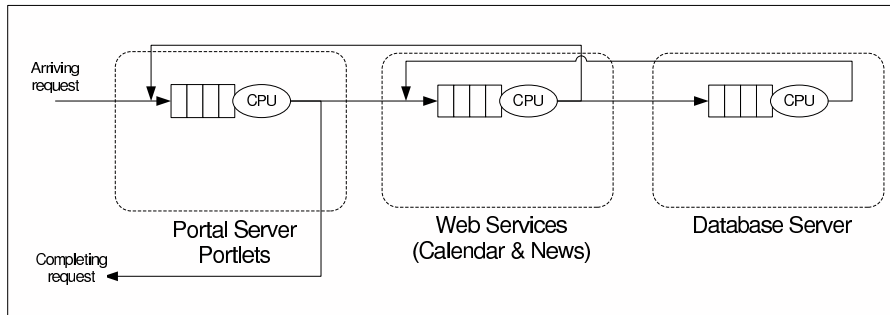


Fig. 2. 3-tiered Portal Application

system that we evaluated. The three CPUs and queues represent the devices for the QN model that we solved.

#### 4.1 Performance Testing

All tests were run on three desktop-class computers running RedHat Fedora Core4. The database server was a 1Ghz Athlon with 512M of RAM, the web services server was a 1.7Ghz Pentium with 1G of RAM, and the portal server was a 2.6Ghz Pentium with 2G of RAM. The operating systems were not tuned in any way and other applications were running at the same time (standard desktop environment) but care was taken not to use the machine while the tests were underway. Apache Benchmark [13] was used to load test the portal container. This program allows you to configure either the number of requests it will make or the time spent on the test and the number of concurrent requests it will make at a given time. The response data was returned with the median response time, the mean response time and breakdown of what percentage of requests were returned within a given response time.

In order to bypass the HTTP-session based user authentication mechanism of Jetspeed, some extra parameters were passed along with the URL of the portal web page. These extra parameters were read by the outermost timing filter and used to populate the HTTP session with the user class and current service level that the QoSWrappers should use. These two parameters were used in order to be able to control the tests in a way that would have been impossible if the QoS monitor system was given full control of the differentiated service levels.

#### 4.2 Portal Configuration and Resolutions

Three resolutions were defined for each of the portlets in the system. The services include a calendar service (for displaying scheduled events in a calendar), event service (for scheduling events in a calendar), and a news service (for displaying top news items). Each utilizes three QoS levels corresponding to high resolution, middle resolution, and low or minimum resolution data.

**Calendar and Event Creation Portlets.** The calendar rendering portlet displays a calendar of events. It communicates with a calendar service that maintains the event list on a per-organization basis in a database. The calendar can also support user-defined events. The service can return any arbitrary date range of events; this feature was used for the differentiated services with no modifications to the calendar service. At the full resolution the calendar displays one month of events, at the mid resolution it displays one week of events and at the low resolution it displays one day of events.

These three levels of resolution impact both the calendar Web service and the portlet computation time. With a populated calendar, the web service needs fewer database queries and less processing to respond to a request for fewer days. As the number of days is minimized, the portlet requires less processing time to loop through the number of days being displayed and render the HTML version of the calendar. While the total time is not impacted greatly in the portlet, the amount of memory used is, and the less memory you use per request in a Java Web server, the less often the memory garbage collection has to run, improving the overall performance of the web server.

The other calendar portlet is an event creation portlet. This portlet allows the user to create a new calendar event, which is processed by the portlet and then sent to the calendar web service. Depending on the user class and the current service level it may be either rendered or disabled.

**News Portlet.** The news portlet follows a similar architecture to that of the calendar. It talks to a news web service that stores the news items in the database and will return a requested number of news items for a given course. The three differentiated service levels for the news portlet follow the same approach as the calendar. For the full level of service, it renders the 10 most recent news items, for the mid level of service, it only renders the 3 most recent items and for the low level of service it only renders the most recent news item.

### 4.3 QN Model and Performance Testing Results

Using the Service Demand Law the service demands  $D_{i,c}$  were computed for each device  $i$  and resolution  $c$  “as the average, for all requests, of the sum of the service times of that resource” [10]. The observed service demands are shown in Table 1. The impact of lowering the resolution is clearly visible in the service demands on the portal server and web services CPUs. The database was not taxed in this model; we believe this is because the data set was relatively small and

**Table 1.** Service Demands

	Portal CPU	Web Service CPU	Database CPU
Full Resolution	0.270	0.225	0.009
Mid Resolution	0.188	0.050	0.004
Low Resolution	0.171	0.018	0.004

could fit in memory. As the resolution is lowered, the service demand on the web services CPU drops drastically; we believe this is because much less data is being requested from the database, processed and converted into SOAP messages. Finally, the service demand on the portal server CPU remains high even as the resolution is lowered, due to the high per-request overhead in Jetspeed. Based on this initial investigation, the resolution factors at low resolution were 0.63 for the portal CPU device and 0.08 for the Web service CPU device.

Once the service demands were computed for the three resolutions, the QN model was solved with 3 seconds as the target response time. The first set of columns in Table 2 shows the predicted throughput of the portal system. The second set of columns shows the results of the performance testing. Performance tests were run for 60 seconds. The concurrency level was changed for each test until a 3 second response time was attained.

**Table 2.** QN Model and Performance Testing Results

	QN Model		Performance Tests	
	Req/Sec	Response Time	Req/Sec	Response Time
Full Resolution	3.52	3.01sec	4.33	2.99sec
Mid Resolution	4.98	3.01sec	5.11	3.03sec
Low Resolution	5.52	3.07sec	5.57	3.05sec

As one can see, the QN model we used did not perfectly predict the throughput necessary to create a 3 second response time. This may be because other programs impacted the measurement of the service demands. While the predicted QN model numbers are not exactly the same, they are quite close and follow the trend of the actual response times and throughput measured during the performance testing phase. The results of both the performance tests and QN model show that the self-healing system will, in this case, allow for a 33% increase in throughput without any changes to the hardware configuration and without reverting to refusing requests. When these tests were run on a single server-class multiprocessor computer the increase was 87%. We presented the results of the slower multiple server scenario to better show the QN model (the multiple CPUs on the server-class computer would be represented by a single device).

## 5 Related Work

Pradhan [14] took the approach of using the request file type as the criterion used to separate the requests into different queues. Each file type queue was assigned a weight, and as the load increased on the server, certain file types were given less priority. By doing so Pradhan shows that observation-based adaptation of the queues is advantageous compared to statically setting the QoS parameters. Uргаonkar [15] and others define and assign requests into multiple user classes to differentiate the service level per request. Their approaches classify the user

class of the request and assign it to the appropriate queue. If the server approaches overload, the lower class requests are dropped or delayed in order to allow the higher user class requests to go through. Menasce [3] modifies the single request queue of the Apache web server in order to balance throughput and request times. By reducing the size of the incoming request queue, he limits the number of concurrent requests the server has to handle, thereby keeping the response time per request under a specified value. The trade-off here is that when the queue is shorter there are more rejected requests. Zhou [16] has a similar user class queuing approach, but also allows for lower class users to enter into the higher class queues if that will not impact the overall QoS of the higher classes. Urgaonkar et al [17] also describe a performance model for multi-tier dynamic websites that uses the Mean-Value Analysis algorithm for closed-queuing networks to estimate response times. They present two solutions to overload situations: dynamic capacity provisioning in order to respond to peak workloads without denying requests and policing requests with an admission control policy that refuses requests that would exceed the SLA.

Research into self-healing web service systems has included self-managing and self-recovering autonomic systems. The majority of the relevant self-managing systems used an autonomic manager to choose between equivalent services. Sadjadi et al. [18] address self-management of composite systems using autonomic computing. Their goal is to use two different equivalent web services (images from a surveillance system) in order to create a fault-tolerant system. Liao et al. [19] also use autonomic computing to manage composition of web services. They use a “federated multi-agent system for autonomic management of web services ... for autonomic service discovery, negotiation, and cooperation”. They propose that the use of autonomic computing to manage the federation of agents will “simplify the control of web services composition, sharing and interaction”. They offer some rules for selecting alternative web services in the case where the currently selected web service is no longer responding within its SLA. Maximilien [20] uses the term “self-adjusting” to describe the mechanism by which web service selection should be undertaken. Other research into self-healing systems has covered the total server failure scenarios [21] and transaction-based models for recovery of failed systems [22].

## 6 Conclusions and Future Investigations

The self-healing portal system presented in this paper is our initial investigation into autonomic systems. The system uses the ability of the underlying services to respond to requests at different resolutions in order to complete the end-user’s request without breaking the SLA. As the load on the system increases, the responses become smaller, until they contain only the most critical information. We also presented a QN model with an added resolution factor to predict the response times and throughput of the portal application.

Using our framework, few changes are necessary in order to convert an existing web application system into a self-healing system. Either the underlying

services are unchanged or a light-weight wrapper that will know how to respond to lower resolution requests needs to be written. Minor changes may be needed to the front-end components in order to render the lower resolution requests correctly. The benefits of the system are evidenced by increased throughput without increased response times. While this implementation does not cover some of the more familiar self-healing functionality (e.g., complete failure of a component or system), we do not foresee anything in the design or implementation that would hamper the co-existence of our self-healing system with other autonomic systems.

We plan on continuing this line of research by implementing our framework in existing open source web applications such as a blogging site, a discussion forum and an e-commerce site. Using the results of this work we will refine our set of resolution factors. We would like to be able to analyze an existing web application and accurately predict, using our knowledge base, the potential impact of the self-healing framework. Future investigations will cover a more robust autonomic manager and integration of other autonomic features and frameworks.

## References

1. N. Feamster, J. Winick, and J. Rexford. A model of bgp routing for network engineering. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 331–342, New York, NY, USA, 2004. ACM Press.
2. Virgilio A.F. Almeida and Daniel A. Menasce. Capacity planning: An essential tool for managing web services. *IT Professional*, 4:33 – 38, Jul/Aug 2002.
3. Daniel Menasce and Mohamed Bennani. On the use of performance models to design self-managing computer systems. In *Computer Measurement Group*, 2003.
4. Vikram Kanodia and Edward Knightly. Ensuring latency targets in multiclass web servers. *IEEE Transactions on Parallel and Distributed Systems*, 14:84–93, 2003.
5. Network world: Autonomic computing. <http://www.networkworld.com/links/Encyclopedia/A/842.html>, November 2002.
6. A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. Technical report, IBM, 2003.
7. A. Zeid and S. Gurguis. Towards autonomic web services. In *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on*, pages 69–, 2005.
8. Paolo Bellavista, Antonio Corradi, Rebecca Montanari, and Cesare Stefanelli. An active middleware to control QoS level of multimedia services. In *IEEE Workshop on Future Trends of Distributed Computing Systems*, page IEEE, 2001.
9. Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002.
10. Daniel A. Menasce, Virgilio A.F. Almeida, and Lawrence W. Dowdy. *Performance by Design*. Prentice Hall, 2004.
11. Daniel A. Menasce, Honglei Ruan, and Hassan Gomma. A framework for qos-aware software components. In *Proceedings of the fourth international workshop on Software and performance*, pages 186–196. ACM Press, 2004.

12. Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 297–304. ACM Press, 2000.
13. Apache Software Foundation. Apache benchmark. [Online] Available <http://httpd.apache.org/docs/programs/ab.html>, March 2005.
14. P. Pradhan, R. Tewari, S. Sahu, C. Chandra, and P. Shenoy. An observation-based approach towards self-managing web servers. International Workshop on Quality of Service, 2002.
15. Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: policing extreme overloads in internet applications. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 740–749, New York, NY, USA, 2005. ACM Press.
16. X. Zhou, Y. Cai, and G. Godavari. An adaptive process allocation strategy for proportional responsiveness differentiation on web servers. In *IEEE International Conference on Web Services ICWS 2004*, pages 142–149, 2004.
17. Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, New York, NY, USA, 2005. ACM Press.
18. S.M. Sadjadi and P.K. McKinley. Using transparent shaping and web services to support self-management of composite systems. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 76–87, 2005.
19. Bei-Shui Liao, Ji Gao, Jun Hu, and Jiu-Jun Chen. A federated multi-agent system: autonomic control of web services. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, volume 1, pages 1–6 vol.1, 2004.
20. E. Michael Maximilien and Munindar P. Singh. Toward autonomic web services trust and selection. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 212–221, New York, NY, USA, 2004. ACM Press.
21. G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr: an autonomous self-recovering application server. In *Autonomic Computing Workshop, 2003*, pages 168–177, 2003.
22. G. Eddon and S. Reiss. Myrrh: A transaction-based model for autonomic recovery. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 315–325, 2005.