

A Self-Healing Framework for Web Services

Henri Naccache

Dept. of Computer Science & Engineering
Arizona State University - Tempe
Box 878809, Tempe, AZ 85287-8809
henri@asu.edu

Gerald C. Gannod*†

Dept. of Computer Science & Systems Analysis
Miami University, Oxford OH 45056
gannodg@muohio.edu

Abstract

Ajax-based web applications are designed to mimic more traditional desktop applications and require quick response times from the underlying Web services. However, since availability and performance of Web services cannot be guaranteed, response time and overall performance of Ajax-based applications can vary. In this paper we describe a framework for developing autonomic self-healing Web service-based applications that rely on the notion of differentiated services (i.e., services that provide common behavior with variable quality of service) in order to maintain required performance characteristics. We present the expected impact of the framework through the use of a theoretical QN model, demonstrate the framework with an example, and provide an evaluation of the technique.

1 Introduction

In modern Ajax-based [10] web applications, the underlying requests made from a browser-based application are often to web services. Ajax-based web applications attempt to mimic the feel of rich client applications by making small requests for data from a web server and, in response, updating a small portion of the web page rather than refreshing the complete page. In order to maintain the feel of a rich client application, the response times of the requests must be small.

In our previous work [18] we explored the impact of implementing an autonomic system in web portal applications. The autonomic responsibility was placed on the portal application, rather than the underlying services. Portlets were instructed to render at various resolutions or were completely disabled, depending on the individual portlet response times and system load. The underlying services that the portlets used were not changed, nor were they aware of

any autonomic systems being used. The results, while positive, were not as pronounced as those presented in this paper. This was due to the nature of the portal application we used, and the heavy service demands placed on the system by the portal application compared to those of the portlets.

In this paper, we describe a self-healing framework aimed at improving the performance of Ajax-based applications. For this framework we define a self-healing system as an autonomic web server that can respond to server overload situations without any system administrator interaction. One of the main attractions to Ajax-based systems is the end-user impression of a very responsive and dynamic web page. As the load on a web service increases, and the response times surpass those set forth in a service level agreement (SLA), the end-user impression of a responsive system will be lost. In order to maintain the responsiveness of Ajax-based applications, our self-healing framework directs the underlying adaptive-content aware components to lower their output resolution. The lowering of the output resolution of the components reduces the service demands of the components on the system and allows for higher request loads to be handled within the same SLA. Differentiating services at the web service level is possible with any application that returns a page-able data set. We define a page-able data set to be one where the end user is expected to take one or more actions (e.g., click next) in order to view the complete information. Examples of this include search results, image galleries, news feeds, blog postings, user posted comments and calendars. Along with the preliminary implementation, we present a Queuing Network (QN) analytical model that predicts the expected increase in request load handling that can be gained by implementing the self-healing framework.

While admission control and queue management have been shown to improve throughput [15] and response times [12], the cost incurred is request refusal. We consider request refusal to be the worst experience an end-user can have. Request refusal is particularly bad in Ajax applications because there is no standard way for the application to

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

†Contact Author.

show the user that the request has failed (such as the 408 error code “Request Timeout” with a traditional web browser request). With our self-healing approach, the server can recover from unacceptable increases in response times by reducing the resolution of the web services to 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 an Ajax 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 selection can be based on full resolution requests at an average load level while allowing for peak loads to be handled at a lower resolution and still maintaining the response time SLA.

As stated by Bouch et al. [5], 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. In our sample application the feedback is visible to the end user as changes to the calendar rendering. The dates for which requested data was not returned are grayed out and a tool-tip is available explaining how the user can request the specific data.

The remainder of this paper is organized as follows. Section 2 describes background material on autonomic computing, Ajax, adaptive content and QN models. The self-healing web services 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 self-healing systems, adaptive content, Ajax and QN models.

2.1 Self-healing systems

An autonomic component consists of two integrated parts: a managed element and an autonomic manager. [25] The manager is in charge of maintaining status information about the managed element and making sure that the managed element remains healthy. In our framework the autonomic manager takes the form of a QoS monitor. The managed elements are the existing underlying web services.

Agarwala [1] explores the impact of QoS monitoring in autonomic systems and presented a differentiated service approach to managing the overhead of the monitoring software. The quality of the monitoring is differentiated to suit the needs of the system and to reduce the load generated by the monitoring software. This research complements our work quite nicely, and is an area of the field we have not explored in depth.

Popular approaches towards self-healing systems fall

into two camps: queue management and dynamic web server selection. Queue management includes modifying the queue properties [17], prioritizing the incoming requests [8], and sorting the requests into multiple priority queues [26]. All of the queue management approaches will finally reject a portion of the requests when the load reaches a certain point. Dynamic web service selection has been proposed as a way of solving performance issues with web services. The basic concept is to have a selection of independent web services that offer the equivalent functionality. If the currently selected web service fails to meet the SLA then another one is seamlessly selected to respond to requests [21]. Our research applies differentiated services techniques to the web services in order not to refuse requests or rely on other web services.

2.2 Adaptive content

Our prior research [18] explored the impact of a differentiated services approach to a web portal application. The system architecture in that case was a more traditional three tiered model where the first tier was the portal application and encapsulated portlet instances. In that system the differentiation took place at the portal application tier, all changes to the resolution of requests to the second tier were handled by the portlet wrappers, and the underlying web services were unaware of any autonomic actions taking place.

The common approach is to re-encode multimedia files into lower-quality, and therefore smaller, files. This has been proposed by Bellavista [4] among others. The majority of that 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. In contrast, our approach deals with adapting the demands put on the server hardware, rather than the network.

2.3 Ajax

Ajax is a set of web and browser technologies that are used to create applications that look and feel more like desktop applications. Ajax technologies are being used to build web applications with more responsive user interfaces, improved performance and higher levels of interactivity [19]. The core technologies behind Ajax are: dynamic HTML (DHTML), cascading stylesheets (CSS), JavaScript, and XML.

The term Ajax was coined by Garrett in early 2005 [10]. While the underlying technologies have been around for much longer than that, and some prominent sites, such as Microsoft’s Outlook web client, have been using it for a few years. The implementation that really ignited the more general usage of Ajax technologies was Google Maps. The ability to drag the map around the screen instead of having to reload the entire page whenever the user needed to pan the map was revolutionary in the web application world.

The fundamental change between standard web applica-

tions and Ajax-driven applications is that standard web applications reload the complete page when any user action takes place. Ajax applications will send an HTTP request in the background to retrieve the requested information and will then update a section of the current web page. If this background request is done fast enough and the update is small enough the impression given to the end user is that of a more responsive dynamic page that “will update immediately without reloading” [19].

The functionality that makes this possible is a JavaScript method called “XMLHttpRequest”. Due to security constraints this JavaScript method can only make HTTP requests to the same server that served the original web page, but is able to retrieve any sort of data that would be sent to a standard browser. On the server side, the web server can not differentiate between an XMLHttpRequest and a standard browser request.

Our test case uses Ajax technologies to make asynchronous requests to a web service for a personal calendar web application. The web service responds to date range requests with a list of summaries of events that will take place within that date range. This type of web application interface is more responsive than a traditional web page [22].

2.4 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 measured 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 [16] 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. We model web services that output multiple resolutions as multiclass systems, with each resolution assigned a request class. 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

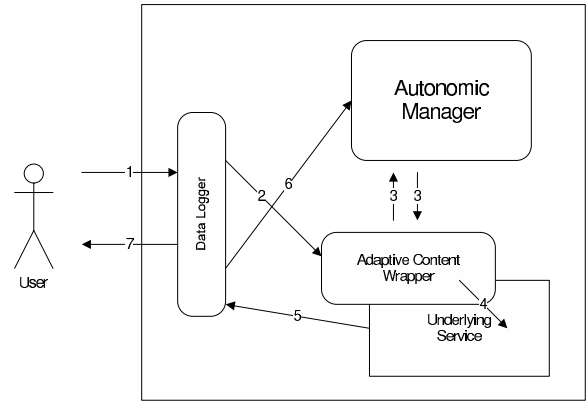


Figure 1. Self-Healing Framework

the system.

3 Approach

This section describes the proposed framework, and presents a QN model with resolution factors for predicting performance.

3.1 Self-Healing Framework

The self-healing framework presented in this paper provides a way to support autonomic webservices by using Ajax technologies. One of the fundamental advantages to using Ajax for websites is the ability to update a portion of a web page without having to reload the whole page. When the autonomic Web services respond to a request at a lower resolution the user is given feedback allowing them to request any data that was not returned in the original request.

In order for an autonomic Web service 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 requests are modified in order to maintain adequate response times. Our current response time monitoring approach uses a simple average of the last N number of requests with a mechanism to handle periods of inactivity. The autonomic manager uses the response times, the load on the server and the system upgrade and downgrade policies to decide what resolution a service should respond to a request with.

The framework is made up of a small set of components, as shown in Figure 1. The Autonomic Manager tracks the current state of the server, the average response times of the services at different resolutions and the resolution at which requests should be handled. The Data Logger calculates response times and sends that information along with other request-specific information to the Autonomic Manager. The Adaptive Content Wrappers are simple filters, written for each existing service, that modify requests depending on the resolution that should be used. For each application only short Adaptive Content Wrappers need to be written.

A request that enters the system is intercepted by our framework and follows the path shown in Figure 1. The steps are as follows:

- (1) A request enters the system.
- (2) The data logger intercepts the request, captures the request arrival time stamp and forwards the request, based on the request type, to the correct adaptive content wrapper.
- (3) The adaptive content wrapper queries the autonomic manager with the request type and the user information. Based on the current state of the system, the autonomic manager responds with the resolution level that the request should be processed at.
- (4) The adaptive content wrapper modifies the request according to the resolution set by the autonomic manager and forwards the new request to the underlying service.
- (5) The underlying service processes the request and returns the results.
- (6) The adaptive content wrapper augments the response with information regarding any differentiation that took place.
- (7) The data logger captures the elapsed request processing time and updates the autonomic manager with the request type and the response time.
- (8) The response is returned to the end user.

3.2 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 our 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 be reduced by some factor.

We represent the impact of lowering the resolution as the resolution factor $F_{i,c,x}$ per user class, where x is the resolution level and X is the total number of resolutions. For a full resolution request $F_{i,c,x} = 1$. 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. (1).

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

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 for all resolutions. The per class utilization is a product of arrival rates and service demands.

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

where λ is the arrival rate of requests of the class c at a resolution x . The first summation in Eq. (2) 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.

One of our goals is to be able to correctly estimate the resolution factors that will 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. We feel that, with enough analysis of the impact of our framework on existing applications, we will be able to make informed estimates based on the type of implementation model and the nature of the data set we are presented with.

4 Evaluation

In this section we describe the process by which we evaluated the self-healing Web service system.

4.1 Self-Healing Monitor Configuration and Resolutions

Our targeted Web service for this evaluation is a personalized calendar service that maintains the event list on a per-user 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 underlying 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 workload and the size of the data set returned from the database. With a populated calendar, the web service needs fewer database queries and less processing to respond to a request for fewer days. The processing requirements, as implemented, are mostly due to the evaluation of recurring requests. Also, as the returned data set decreases in size the amount of memory used for temporary objects is lower, 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 autonomic manager collects performance statistics and maintains a set of QoS parameters. The main parameter is the average response time for a given class of request. This response time is calculated as an average of the last N requests, where N is a configurable option that was set to 50 for our tests, and takes into account gaps in request arrivals in order to re-set the average values during times of inactivity. Based on the current average response times, an incoming request is assigned a differentiation level by the manager. This differentiation level is used by the self-healing calendar web service wrapper to change the date range of the request before it is passed to the calendar web service. When the event list is returned from the calendar web service the self-healing wrapper injects information into the resulting data set to indicate what range of requested days, if any, was not queried. For example, if the request was for a complete month and the level was set for one week's worth of data, the current week of the month was returned, with the rest of the days flagged as not having been examined. For a differentiated level of a single day, the current day of the month was returned. In both cases there is a chance that the user wished to see a different data set. The assumption is that they will then request the specific day or week they are looking for in a second request. This turns the single 1-month request into 2, or potentially more, requests for specific weeks or days. Even in the situation where the user ends up requesting the same total data, it will have been spread out over time among multiple requests which will help alleviate the load on the server.

4.2 QN Model and Performance Testing Results

In order to model our self-healing web service system the devices that will impact the response time of the server need to be identified. For our calendar web service we created a very simple QN model with a single CPU representing the processing unit in the web services server. Disk IO due to database activity is negligible in our test scenario (the data set is small enough to fit into memory) and no files are read in order to respond to a calendar request.

Since the calendar web service was an existing application, we were able to measure the service demand for a standard one month data request. Using the Service Demand Law [16] we measured the number of requests the server responded to over a set period of time and the utilization of the CPU.

$$D = \frac{U * T}{C} \quad (3)$$

The service demand on the web server CPU is calculated using Equation 3. U is the utilization of the cpu, T is the time, in seconds, over which the utilization was measured and C is the number of requests handled by the server during that time period.

	Measured Service Demand D	Resolution Factor	$D * F$
One Month	0.04 sec/req	1	0.04
One Week	-	1 / 4.5	0.009
One Day	-	1 / 30	0.001

Table 1. QN Model Service Demands and Resolution Factors

The resolution factor F can be thought of as an estimation of the impact of lowering the resolution will have on the load on a service. Making an assumption that the relationship between the number of days requested and the service demand imposed on the system would be linear, we defined the resolution factors as seen in Table 1. The factors F were reached by calculating the average number of weeks and days in a month.

The impact of lowering the resolution is clearly visible in the service demands on the Web services CPU. In order to solve the QN model we made another assumption regarding the percentage of requests that would be differentiated to lower resolution levels. Using a split of 30% of requests left at one full month, 30% reduced to one week and 40% to one day we solved the QN model for a 1 second response time. The concurrent request load was calculated to be 64.5 requests/second, which is very close to what was measured in our performance tests.

4.3 Performance Testing

All tests were run on three desktop-class computers running RedHat Fedora Core6. The Web services server was a 1.7Ghz Pentium with 1.5G of RAM. The operating systems were not tuned in any way. A minimal set of applications was running at the same time, but care was taken not to use the machine while the tests were underway. Apache Tomcat 5.5 was used as the Web server and had the maximum number of threads set at 1000 in order not to throttle the incoming requests at the server level. The maximum number of database connections to the Postgresql database server was set at 250 (higher than the number of concurrent requests) in order not to cause database connection pool related starvation to incoming requests.

Apache Benchmark [9] was used to load test the Web services. This program allows you to configure the number of concurrent requests it will make at a given time and either the total number of requests it will make or the time spent on the test. 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. The autonomic manager logs were also analyzed for internal Web service response times, to measure how many requests were differentiated and what level of resolution was returned.

The performance tests were run for 1000 requests at concurrency rates ranging from 5 to 60 parallel requests. Each

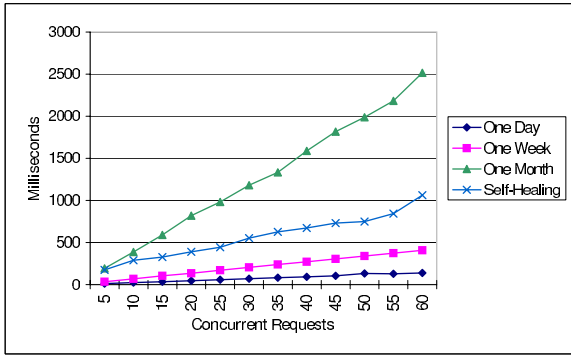


Figure 2. Response Times vs Concurrent Request Load

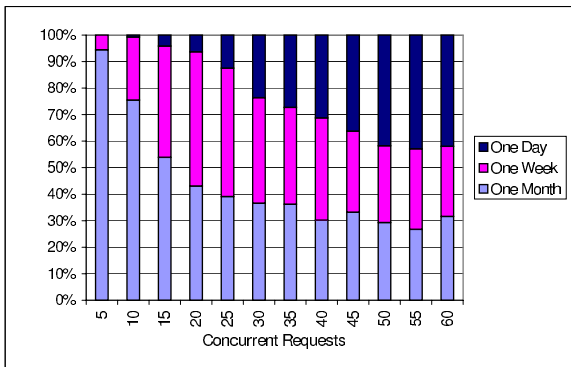


Figure 3. Differentiated Service Levels for each Concurrent Request Load

test was repeated 5 times and the results presented in this paper represent an average values over these 5 tests. In order to create a baseline, the autonomic manager was configured not to change the resolution for the one month, one week and one day plots. The “one week” plot is an average of the 5 weeks in the month we tested. The “one day” plot is an average of each of the 30 days in the month we tested.

Figure 2 shows the response times for the calendar service for each individual resolution and with the self-healing framework in place. The average response times are shown in milliseconds for each level of concurrent requests. As expected, the lower the resolution of the output, the less time it takes to respond to the request. The “one month” plot shows how quickly the default system fails to meet the 1 second SLA we are aiming for. With our self-healing framework in place the response times are kept under 1 second with close to twice the load. An ideal autonomic manager would show a response time plot that would follow the “one month” plot until it reached 1 second and then it would maintain the response times at that level for as long as possible. Future work on the autonomic manager will be directed towards this goal.

The response times should be read with the differentiated

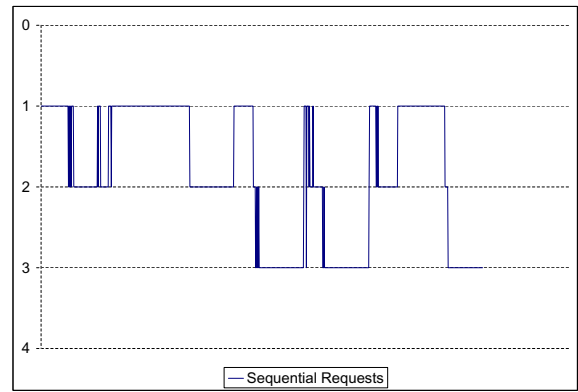


Figure 4. Differentiated Service Levels Per Individual Request (60 Concurrent Requests)

service distribution graph, found in Figure 3, in order to see what percentage of requests were returned at a lower resolution. The graph shows the distribution of requests at each resolution versus the number of concurrent requests. This graph shows how our current autonomic manager handles the increased load. While the autonomic manager could have reduced the percentage of “one month” responses at the higher concurrency rates, we felt that for this implementation a 2x increase in the request load was enough to show the effectiveness of our self-healing framework.

Figure 4 plots the resolutions returned for each of the 1000 requests at a concurrency rate of 60. This is one of the five performance tests used to generate the last entry in Figure 3. A resolution level of “1” is a request handled at “one month”, “2” is a request handled at a “one week” level and “3” is a request handled at a “one day” level. As we can see in the graph, the system was initially able to maintain the SLA with “one month” responses and two bursts of “one week” responses. At the middle of the performance test run, as some of the slower requests are handled and the average response time increases, the autonomic manager instructs the calendar service to drop to “one day” responses. Longer performance tests we have examined show similar patterns of autonomic management. We are currently investigating a more preemptive autonomic manager that will intersperse lower resolution responses earlier in the cycle in order to minimize the large blocks of “one day” responses.

These results show the effectiveness of our self-healing system in keeping the response times below a 1 second cut-off for a much higher load than the original implementation. Our autonomic manager is still a very basic implementation. Ideally we would return full resolution requests and the response time would follow the full month plot until it met a pre-set level (1 second). At that point the response time plot would flatten while the resolution levels would fluctuate much more rapidly in order to maintain the 1 second response time goal.

5 Related Work

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. [21] 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. [13] 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 [14] 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 [6] and transaction-based models for recovery of failed systems [7].

Due to Ajax security restrictions, Web requests can only be placed to the server that responded to the original Javascript file. Therefore, any system that involved dynamic Web service selection would have to include a proxy application running on the original Web server, which would add another layer of Web requests and potential bottlenecks. The other requirement of a system that dynamically selects between distinct Web services is that requests can not contain session information or personalized information if a seamless transition between services is expected.

Angelaccio [3] shows the improvement in performance of a Web-based chat application when it is converted to the Ajax model of Web development. Hanakawa [11] explored the performance impact of using Ajax-like asynchronous Web requests on existing Web sites. They found a distinct performance advantage in using asynchronous requests that generate partial updates of a Web page in situations where the load on the server is high.

Pradhan [20] 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. Urgaonkar [24] 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 the request 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 [15] 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 [27] 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 [23] 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.

6 Conclusions and Future Investigations

The self-healing portal system presented in this paper is a continuation of our initial [18] 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 violating 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 reasonably predict the response times and throughput of the multi-resolution output of the self-healing web services.

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 slightly modified or a light-weight wrapper that will know how to respond to lower resolution requests is written. The client applications will need some modifications in order to handle the varied resolutions that are returned by the web services. 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 to continue 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. 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 that will be more dynamic in the levels of differentiation, integration with the research into differentiated QoS monitoring done by Agarwala [1], and integration of other autonomic features and frameworks.

References

- [1] S. Agarwala, Y. Chen, D. Milojevic, and K. Schwan. Qmon: Qos- and utility-aware monitoring in enterprise systems. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, 2006.
- [2] V. A. Almeida and D. A. Menasce. Capacity planning: An essential tool for managing web services. *IT Professional*, 4:33–38, Jul/Aug 2002.
- [3] M. Angelaccio and B. Buttarazzi. A performance evaluation of asynchronous web interfaces for collaborative web services. In *ISPA 2006 Workshops: Frontiers of High Performance Computing and Networking*, volume 4331/2006 of *Lecture Notes in Computer Science*, pages 864–872. Springer Berlin / Heidelberg, November 2006.
- [4] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. An active middleware to control QoS level of multimedia services. In *IEEE Workshop on Future Trends of Distributed Computing Systems*, page IEEE, 2001.
- [5] A. Bouch, A. Kuchinsky, and N. 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.
- [6] 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.
- [7] 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.
- [8] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 276–286, New York, NY, USA, 2004. ACM Press.
- [9] A. S. Foundation. Apache benchmark. [Online] Available <http://httpd.apache.org/docs/programs/ab.html>, March 2005.
- [10] J. J. Garrett. Ajax: A new approach to web applications. [Online] <http://www.adaptivepath.com/publications/essays/archives/000385.php>, 2005.
- [11] N. Hanakawa and N. Ikemiya. A web browser for ajax approach with asynchronous communication model. In *Web Intelligence, 2006. WI 2006. IEEE/WIC/ACM International Conference on*, pages 808–814, 2006.
- [12] V. Kanodia and E. Knightly. Ensuring latency targets in multiclass web servers. *IEEE Transactions on Parallel and Distributed Systems*, 14:84–93, 2003.
- [13] B.-S. Liao, J. Gao, J. Hu, and J.-J. 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.
- [14] E. M. Maximilien and M. 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.
- [15] D. Menasce and M. Bennani. On the use of performance models to design self-managing computer systems. In *Computer Measurement Group*, 2003.
- [16] D. A. Menasce, V. A. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.
- [17] D. A. Menasce, D. Barbare, and R. Dodge. Preserving QoS of e-commerce sites through self-tuning: a performance model approach. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 224–234. ACM Press, 2001.
- [18] H. Naccache, G. C. Gannod, and K. A. Gary. A self-healing web server using differentiated services. In *ICSOC 2006: 4th International Conference on Service Oriented Computing*, volume 4294/2006 of *Lecture Notes in Computer Science*, pages 203–214, November 2006.
- [19] L. D. Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.
- [20] 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.
- [21] S. Sadjadi and P. 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.
- [22] K. Smith. Simplifying ajax-style web development. *Computer*, 39(5):98–101, 2006.
- [23] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. 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.
- [24] B. Urgaonkar and P. 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.
- [25] 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.
- [26] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *Proceedings of the twelfth international conference on World Wide Web*, pages 411–421. ACM Press, 2003.
- [27] 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.