EFFECTIVE WEB SERVICE LOAD BALANCING THROUGH STATISTICAL MONITORING

By GEORGE PORTER *and* RANDY H. KATZ

Identifying the correlated effects between components to improve response to overload.

Web services are increasingly used for deploying Web-based application portals such as eBay and Amazon.com. They are commonly built using middleware, that is, composable building blocks such as HTTP containers, Java-based application beans, and persistent state management. These components are distributed across tiers of servers—Web, application, and database. As Web services offer newer and more sophisticated functionality, the underlying realization of those services in the middleware becomes more complicated. Today's Web services can consist of dozens or hundreds of request types and middleware components.

Illustration by Jean-François Podevin

WE NEED A MORE SOPHISTICATED WAY OF LOOKING THROUGH THE LARGE AMOUNT OF DATA COLLECTED AT EACH POINT TO DISCERN CORRELATIONS BETWEEN COMPONENTS.

This separation and replication of components allows Web services to scale in response to new resource demands. This is accomplished by introducing new servers hosting the particular, needed component. Despite this scalability, flash traffic patterns can drive a Web system's middleware component (or components) into overload. This leads to poor performance as the system is unable to keep up with the demands placed on it, and users see increased response times for their requests (see Figure 1). Experiments have indicated that users can tolerate roughly eight seconds of delay before they either retry their request or leave the site [2].

While the need for an admission control scheme is clear, formulating an effective system is daunting and error-prone. This is due to the large number of request types and middleware components. Different requests to a Web service stress different middleware components [1, 3]. It is advantageous to preferentially throttle those requests most correlated with the bottleneck. To do that, better visibility into the relationship between requests and their effects is necessary. Unfortunately, current system software and site-monitoring tools do not provide the operator with this visibility. For Web services to be more self-adaptive, they need to be more introspective, identifying correlated effects between internal components, so that the operator can act to shed load from overloaded components without penalizing all users of the Web site.

To design such self-adaptive Web services able to gracefully respond to overload, we propose four design mechanisms: simple statistical techniques for uncovering request effects in multi-tier systems; a black-box approach to middleware component monitoring; a visualization tool summarizing statistical findings to facilitate human decision making; and efficient techniques for operators to invoke admission control decisions based on those findings. We will also argue that including humans in the loop complements, rather than detracts from, self-adaptive design goals. We present ongoing work on a Web service based on the open source RUBiS auction site (see rubis.objectweb.org) that embodies these mechanisms. RUBiS is a Web-service benchmark designed to profile the performance of an auction site like eBay. Our approach leads to a Web service that is able to serve 70% more requests per second. Additionally, the maximum request latency seen by the user is reduced by 78%. These initial results show promise that middleware-based Web services can greatly benefit from more self-adaptive design.

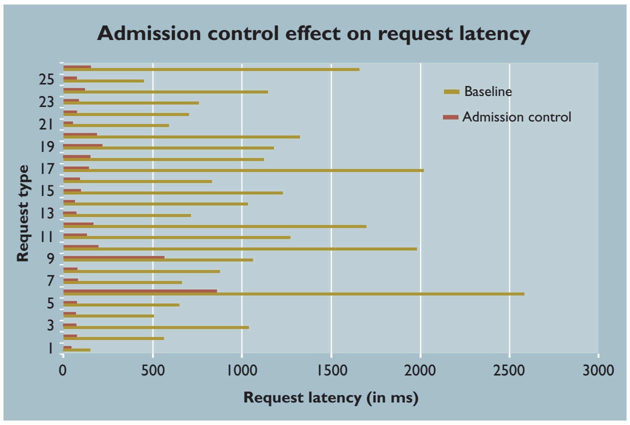

Figure 1. Selectively applying admission control to those requests correlated with the system bottleneck substantially reduces average request latency for the Web service.

Related Work

We can study the performance and operation of Java-based middleware systems using the RUBiS system. The authors of RUBiS showed that the mixture of requests—the workload—plays a significant role in determining system bottlenecks [1, 3].

Our approach differs from previous attempts to apply control theory to operating systems and three-tier systems, which have assumed that requests to the system are homogeneous (affect the system in the same way). The SWIFT system [6, 7] is a systematic approach to introducing input/output interfaces to operating system components, which matches well with the well-defined interfaces between middleware components. The ControlWare system [10] is a toolkit for automatically mapping QoS requirements into simple control loops in three-tier systems.

Considerable work has been applied to correlation analysis of Web services both in research literature and in industrial best practices. The SLIC project at HP Labs [4] attempts to identify which components are responsible for Web service violations of Service-Level Operations (SLOs) by using fine-grained monitoring and instrumentation. The Performance Management project at IBM has explored using control theory and statistical monitoring to detect and adapt to unexpected traffic surges [8, 5]. Techniques for visualizing structured data are described in [9].

Overload Avoidance in Self-Adaptive Web Services

Overload occurs when the load placed on a Web service exceeds its ability to serve requests. Flash traffic and sudden load spikes operate at timescales faster than operators can upgrade their systems. Web service operators can manage load in a number of ways. One way is to direct load to spare servers that can handle the surge. This technique is an example of load balancing. Complex Web services are often built in multiple layers of interconnecting components (see Figure 2). Applying a load-balancing strategy in this environment is non-trivial, since detailed instrumentation of the internal components is usually not available.

High-level overload mitigation strategies can be used, at least temporarily, during this time (such as HTTP 503 TOO BUSY responses). However, this adversely affects all traffic to the site, even when the bottleneck is driven by a small population of requests (about 15%, in our RUBiS emulation). This motivates the desire for a less disruptive, selective admission control.

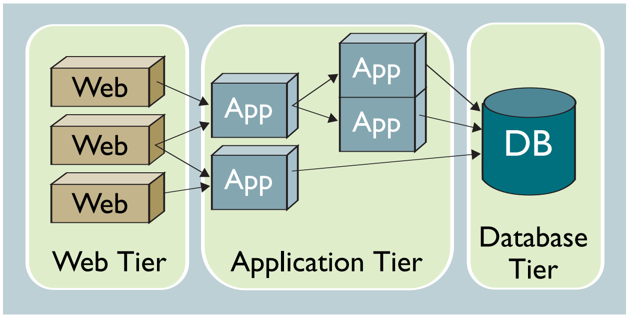

Figure 2. A complex Web service consisting of Web, application, and database components.

In selective admission control, we first throttle back requests contributing to the overload, while leaving all other requests unaffected. In our implementation, the bottleneck was the database's CPU, and the two contributing requests involved searching for items. In general, it is quite difficult to determine the runtime connections between components in a distributed system. Often these are determined by the workload, and can change over time. In addition to the lack of visibility into these connections, it is non-trivial to map those connections from a request to a bottleneck(s). We seek to make use of measurement data in this process.

Problem Statement: Given a system bottleneck component C, identify those requests correlated with C. The data used for that purpose should be collected with minimal disruption to the system. Once identified, reduce the number of correlated requests until the system is no longer overloaded.

We now outline the four mechanisms of our approach in more detail.

Uncovering Request Effect through Correlations

When a request arrives at the Web server, it may invoke processing in one or more Java components in an application tier. In turn, these either access the database or return a result directly to the user. While logging and status information is available on each of the servers hosting these tasks, there are no good system tools for understanding crosscuts through the layers. Given the large number of possible crosscuts, we need a more sophisticated way of looking through the large amount of data collected at each point to discern correlations between components.

To find which requests are correlated with our bottleneck, we make use of the Apache Web logs collected from the Web tier and the CPU load average as reported by the `sysstat` tool. We chose to use Pearson's Correlation Coefficient to find the relationship between request type and CPU load, because it is efficient and simple to calculate, and gives good results in practice. For that statistical technique, we processed the logs as follows:

1. Divide the Apache Web log into time intervals t_{int} (we chose $t_{int} = 1$s).
2. For each interval, count the number of each request type.
3. Form a $n \times m$ matrix M where n is the number of time intervals, and m is the number of request types. Element (i,j) of M represents the number of requests of type j that arrived in time period i.

We then find the correlation between columns of this matrix and the vector of CPU load from the database (this technique can be used for other bottlenecks such as disk I/O). The result is shown in Table 1. The request types highlighted in bold are those significant (to the $\alpha = 0.05$ level) and positive. These are the candidate request types that should undergo admission control. We found the results surprising, since before performing the analysis we expected more of the requests to be correlated with database CPU load, for example BrowseCategories.php (which returns more results than SearchItemsByCategory. php). In fact, the requests identified by our algorithm represent a small fraction of the total requests, yet account for a large load on the database. One issue in large systems is uncovering "uninteresting" correlations. Given a large number of systems metrics, many will be correlated with almost any system component or request workload. These cases can managed by training the model, however this is outside the scope of this work.

Request Type	P-value	Coefficient
BrowseCategories.php	0.1747	-0.035
BrowseRegions.php	0.0926	-0.0434
SearchItemsByCategory.php	**0**	**0.5654**
SearchItemsByRegion.php	**0.0034**	**0.0756**
AboutMe.php	0.7702	0.0075
RegisterUser.php	0.4876	-0.0179
SellItemForm.php	0.4891	0.0179
RegisterItem.php	0.8767	0.004
ViewItem.php	0.0953	-0.0431
PutComment.php	0.5157	-0.0168
ViewUserInfo.php	0.4646	-0.0189
PutBidAuth.php	0.8641	-0.0044
PutBid.php	0.2566	-0.0293
BuyNowAuth.php	0.971	-0.0009
BuyNow.php	0.1206	0.0401
ViewBidHistory.php	0.9741	-0.0008

Table 1. Request effects on the system bottleneck as discovered by Pearson's correlation coefficient. Highlighted entries are statistically significant and have positive correlations. We choose these requests as candidates for selective admission control.

Black-Box Component Monitoring

Self-adaptive systems rely on sufficient self-monitoring to drive statistical inference algorithms, while monitoring should be as minimally invasive as possible. There are at least three motivations driving this requirement:

- ***Fear of disrupting a running service.*** When we discussed implementing our approach on a large, political Web log, the operator responded by saying: "My concern, obviously, is that (the site) isn't a laboratory project, but a real-world application that must maintain as high an uptime as possible. So I'd be wary of experimenting in a way that would potentially compromise service." Thus, we based our statistical analysis on data that was easily accessible and routinely collected (Web logs and `sysstat` measurements). Additionally, instrumenting operating systems components like file systems and system call interfaces is very system-specific and requires expert knowledge. Since hardware and software changes are often frequent events, such low-level instrumentation code would need to be rewritten each time a component is upgraded or changed.
- ***Rapidly changing services.*** A fact of the Web today is that it undergoes rapid changes: the capabilities of the site change, as well as the patterns of traffic arriving at the site. A large Web site likely upgrades hardware and software components on a regular basis. Operators will resist invasive monitoring and instrumentation that must be replicated whenever system components are upgraded. By treating each component as a black box, we do not modify individual system components (such as the file system, operating system calls, or other hooks). This makes our approach more portable as well as less invasive.
- ***Distributed ownership of components.*** Depending on the nature of the service, responsibility for the site might be partitioned between several system operators. Coordinating monitoring operations between these can be difficult. By focusing on high-level component monitoring, different groups do not have to coordinate software upgrades and system modifications. Additionally, it may be impossible to instrument components that are not open source.

As our results will show, high-level measurements are often sufficient for identifying correlations and request effects that can greatly improve running systems.

A Visualization Tool for Automatic Overload Mitigation

We advocate an approach for building self-adaptive Web services in which the operator plays an important role and remains "in the loop." By better visualizing underlying connections between components and load, we claim that operators can become better decision makers. An example of the type of visualization we advocate is given in Figure 3, in which the pie chart shows the percentage of traffic that is correlated to our bottleneck. Within the correlated slice, specific request types are enumerated. From this simple graph, an operator can see which requests would be affected by selective admission control, as well as what percentage of the total traffic they represent.

Revising the three motivations from the previous section, we see that the visualization component reduces the disruption fear by providing a point for the operator to see information needed to diagnose and pinpoint performance problems. Once operators feel more comfortable with the tool, it can be made more automatic. Secondly, to cope with rapidly changing services, visualization tools allow operators to choose whether to implement throttling depending on formal or informal business rules that are known to the operator. Again, as the tool is used more often, some admission control decisions might be programmed to take effect automatically without operator involvement. Lastly, we address the distributed ownership of components. By visualizing request effect through the system, observations across different components (often in different parts of the enterprise) can be correlated into one display that gives more insight to the system's operation.

Effective Actuators for Admission Control

Once a subset of the requests are identified as candidates for selective admission control, the operator needs a way to reduce the rate at which they arrive. This can be done at the HTTP level through 503 TOO BUSY status messages, or at the network level through bandwidth shaping. We chose to implement the throttling at the network level because that did not involve modifying the Web tier. Correlated requests were sent over a bandwidth-limited network path. The effect on the end user for such requests is they take longer than they used to. This means that sessions, which consist of multiple, individual requests, might take longer than before.

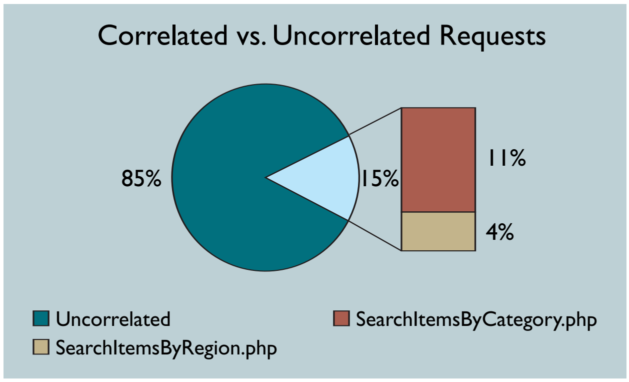

Figure 3. This visualization shows the requests identified by our system as candidates for selective admission control. Additionally, the graph shows their percentage of the total number of requests.

To tie together the visualization tool and the actuators for admission control, we envision an interface in which each request type is listed, along with its likelihood of relieving the noticed bottleneck based on our statistical findings. Such a display resembles a "top talkers" graph. In Figure 3, they would be able to select requests identified by the bar graph. Once selected, new filters could be sent to the Web server (in the case of HTTP-based throttling), or to the network appliance (for network-level throttling). In either case, the operator would have a tactile way to see the effect of their choice on both the bottleneck and the arriving traffic.

Scenario	Total Requests	Correlated URLs	Requests/ Second	Average Session Time (s)	Maximum Request Time (s)
Stock	756,137	112,521 (14.9%)	462	670 s	154.7 s
Selective Admission Control	1,143,264	105,964 (9.3%)	782	872 s	32.7 s

Table 2. Performance measurements for a stock deployment and one that utilizes selective admission control. Both measurements represent 30 minutes of elapsed time with 3,500 concurrent clients. A session represents a series of operations on the auction site.

Results

We have deployed a Web service based on the RUBiS auction site that embodies the four mechanisms outlined in this article. Our testbed consists of an IBM BladeCenter with two Nortel Layer 2-7 Gigabit switches. The Nortel switches can perform deep packet inspection to identify HTTP request types (based on URL pattern matching) at gigabit rates and assign VLAN tags to packets that should be subject to admission control. To perform the bandwidth throttling, we use the Linux Traffic Control (`tc`) extensions to reduce the rate of correlated requests from 3.5Mbit/s (the baseline rate) to 1Mbit/s.

As Figure 1 and Table 2 show, performing this selective admission control greatly improves the performance of the Web service for users who are not causing bottlenecks. In our deployment, the number

SELF-ADAPTIVE SYSTEMS RELY ON SUFFICIENT SELF-MONITORING TO DRIVE STATISTICAL INFERENCE ALGORITHMS, WHILE MONITORING SHOULD BE AS MINIMALLY INVASIVE AS POSSIBLE.

of requests per second went from 462 to 782. This gain is possible because the number of heavy requests (those correlated with the bottleneck) allowed per unit time is reduced. Therefore, it will take longer to search for a series of several items. This is highlighted in the longer average session time (872 seconds vs. 670 seconds), as each session consists of a set of subsequent searches, among other operations. As demonstrated by the positive effect on the number of requests per second and the maximum request time, such a reduction provides a great benefit for many visitors of the Web site.

Conclusion

We have proposed an approach to building self-adaptive Web services based on four design mechanisms: simple statistical techniques for uncovering request effects in multi-tier systems; a black-box approach to component monitoring; a visualization tool for summarizing statistical findings; and efficient techniques for invoking admission control decisions. We are in the process of building an auction Web service embodying these mechanisms, and preliminary results are promising: we achieved a 70% increase in the number of pages served per second, and a 78% decrease in the maximum latency seen by users accessing the Web site. We are encouraged by these results, as they show the promise in building and deploying more self-adaptive Web services. **C**

References

1. Amza, C., Cecchet, E., Chanda, A., et al. Specification and implementation of dynamic Web site benchmarks. In *Proceedings of IEEE 5th Annual Workshop on Workload Characterization* (WWC-5), (Austin, TX, Nov. 2002).
2. Bhatti, N., Bouch, A., and Kuchinsky, A. Integrating user-perceived quality into Web server design. In *Proceedings of the 9th International World Wide Web Conference (WWW9)*, (Amsterdam, The Netherlands, May 2000).
3. Cecchet, E., Chanda, A., Elnikety, S., et al. Performance comparison of middleware architectures for generating dynamic Web content. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, (Rio de Janeiro, Brazil, June 2003).
4. Cohen, I., Chase, J.S., Goldszmidt, M., et al. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, (San Francisco, CA, Dec. 2004).
5. Diao, Y., Lui, X., Froehlich, S., et al. On-line response time optimization of an Apache Web server. In *Proceedings of the 11th International Workshop on Quality of Service (IWQoS '03)*, (Monterey, CA, June 2003).
6. Goel, A., Steere, D., Pu, C., et al. Adaptive resource management via modular feedback control, 1999.
7. Goel, A., Steere, D., Pu, C., et al. Swift: A feedback control and dynamic reconfiguration toolkit. Technical Report CSE-98-009, Oregon Graduate Institute, Portland, OR, June 1998.
8. Lassettre, E., Coleman, D.W., Diao, Y., et al. Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2003)*, (Heidelberg, Germany, Oct 2003). LNCS, Vol. 2867, Springer, 82–92.
9. Tufte, E. *Envisioning Information*. Graphics Press, 1990.
10. Zhang, R., Lu, C., Abdelzaher, T., and Stankovic, J. Controlware: A middleware architecture for feedback control of software performance. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, (Vienna, Austria, July 2002).

George Porter (gporter@cs.berkeley.edu) is a doctoral student in Computer Science at the University of California, Berkeley.
Randy H. Katz (randy@cs.berkeley.edu) has been a faculty member in Computer Science at UC Berkeley since 1983, where he is now the United Microelectronics Corporation Distinguished Professor.