Adaptive Resource-based Web Server Admission Control

Thiemo Voigt
Swedish Institute of Computer Science
thiemo@sics.se

Per Gunningberg
Uppsala University
Department of Information Technology
Per.Gunningberg@it.uu.se

Abstract

Web servers must be protected from overload since server overload leads to low server throughput and increased response times experienced by the clients. Server overload occurs when one or more server resources are overutilized. In this paper we present an adaptive architecture that performs admission control based on the expected resource consumption of requests. By dynamically setting the maximum rate of accepted requests, we avoid overutilization of the critical resources. We also provide mechanisms for service differentiation. We present our admission control architecture and experiments that show that it can sustain low response times and high throughput for premium clients even during high load.

1 Introduction

Many web server architectures deploy admission control to maintain high throughput and low response times during periods of peak server load. Servers become overloaded when one or more critical resources – CPU, disk and network interface – are overutilized. Depending on the current workload, some server resources can be overutilized, while the demand on other resources is not very high because certain types of requests utilize one resource more than others. This paper presents an architecture that avoids overutilization of individual server resources by taking into account the resource consumption of requests when performing admission control. Most web server architectures reject excess requests without discriminating between different resource bottlenecks [3, 4] or use only one indicator for overload, often CPU utilization [5]. Qguard [8] bases the admission decision on network-level information such as IP addresses and port numbers. Hence, they cannot take the potential resource consumption of requests into account, but have to reduce the acceptance rate of all requests when one resource is overutilized.

In our architecture individual server resources are protected from overutilization by dynamically setting the acceptance rate of resource-intensive requests. Resource-intensive requests and the resource they demand are identified by the URL in the HTTP header. We supervise the resources CPU and network interface and adapt the rate of CPU and bandwidth intensive requests according to the utilization of the corresponding resource. The adaptation of the acceptance rates is done using feedback control loops. Techniques from control theory have been used successfully in server systems before [9, 6, 4].

The main focus of this paper is the provision of service differentiation in our architecture. In our adaptive architecture accepted requests can be processed quickly since server resources are not overutilized. This leads to low response times. Therefore, the major goal of service differentiation in our architecture is to provide high throughput to premium clients. This is done by splitting the token buckets used for admission control into logical partitions [10]. Each logical partition corresponds to one service class with larger partitions for more important service classes.

We have implemented this admission control architecture in the Linux OS. Our results show higher throughput and much lower response times during overload compared to a standard Apache on Linux configuration. Also, service differentiation works as expected.

The rest of the paper is structured as follows: The next section presents the system architecture and Section 3 illustrates how the architecture provides service differentiation. Section 4 presents experiments that evaluate various aspects of our system. We conclude after a brief discussion.

2 Architecture

Our admission control architecture deploys two mechanisms that have been described earlier [12].

- *TCP SYN policing* limits acceptance of new SYN packets based on compliance with a token bucket policer. Token buckets have a token rate, which denotes the average number of requests accepted per second and a bucket size which denotes the maximum number of requests accepted at one time. TCP SYN policing enables service differentiation based on information in the TCP and IP headers of the connection request (i.e, the IP source and destination addresses and port numbers).

- *HTTP header-based connection control* is activated when the HTTP header is received. Using this mechanism a more informed control is possible which provides the ability to, for example, specify lower access rates for resource-intensive requests. This is done using filter rules, e.g., checking URL, name and type.

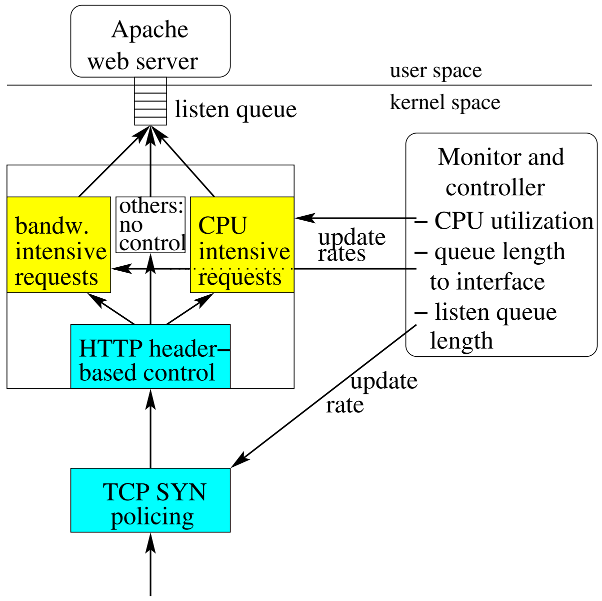


Figure 1. Kernel-based architecture

The admission control architecture is depicted in Figure 1. It contains the following entities:

- A TCP SYN policer.
- An HTTP header-based connection control entity to identify and rate control resource-intensive requests.
- A resource monitor and control module.
- An unmodified Apache web server.

Our architecture assumes that resource-intensive web objects such as CGI-scripts are grouped according to their resource demand in the web server's directory tree. For example, all CPU-intensive scripts should reside in the `/cgi-bin` directory. This way, HTTP header-based connection control using a filter rule `/cgi-bin` and the associated token bucket policer is able to restrict the acceptance of CPU-intensive requests.

On receipt of a request, HTTP header-based connection control parses the URL and matches it against the filter rules. If there is a match, the corresponding token bucket is checked for compliance. Compliant requests are inserted into the listen queue, non-compliant requests are discarded by resetting the connection[1]. We call this part of our admission control architecture *resource-based admission control*. Resource-based admission control makes sure that resource-intensive requests do not cause overutilization of the corresponding critical resources. We supervise the resources CPU and network interface. Both high CPU utilization and dropped packets on the networking interface can lead to long delays and low throughput. Other resources that could be controlled are disk I/O bandwidth and memory.

As Figure 1 shows, we do not perform resource-based admission control on all requests. Requests such as those for small static files do not put significant load on one resource. However, if requested at a sufficiently high rate, these requests can still cause server overload. Hence, admission control for these requests is needed. It would have been possible to insert a default rule and use another token bucket for these requests. Instead, we have decided to use TCP SYN policing and police all incoming requests.

The main reason for deploying TCP SYN policing is its early discard capability. Using SYN policing, less resources are wasted for requests that are eventually discarded. One of our design goals is to keep TCP SYN policing inactive while resource-based admission control can protect resources from being overutilized. Resource-based admission control can access other application-level information, such as cookies. This enables the identification of ongoing sessions or premium clients while SYN policing cannot access such information.

For each of the critical resources, we use a feedback control loop to adapt the token rate at which we accept requests demanding that particular resource. The adaptation of the rates is done using proportional and derivative controllers [7]. We do not adapt the bucket size but assume it to be fixed.

We deploy one controller, the CPU controller, to adapt the acceptance rate of CPU-intensive requests based on the current CPU utilization. The bandwidth controller adapts the acceptance rate of large, static requests based on the length of the queue to the network interface. In addition, we use a third controller that is not responsible for a specific resource but performs admission control on all requests, including those that are not associated with a specific resource. The latter controller, the SYN controller, is responsible for the rate of the TCP SYN policer. The rate of the SYN controller is adapted based on the length of the listen

[1]A nicer solution is to send a "server temporarily unavailable" (503 response code) back to the client and close the connection.

queue.

Both HTTP header-based connection control and TCP SYN policing are located in the kernel to avoid a context switch to user space for rejected requests. The deployment of mechanisms in the kernel has proven to be much more efficient and scalable than in the web server [12]. Note that usually connections are enqueued into the server's listen queue before the HTTP header is received. In our architecture we delay this enqueuing until the HTTP header is received, parsed and HTTP header-based connection control has been performed.

3 Service Differentiation

One of the design goals of our architecture is to provide better service to premium clients, for example, clients having a service contract with the service provider. In an adaptive architecture the length of the listen queue is almost always zero. This means that traditional mechanisms to provide service differentiation in web servers, such as having different queues for each service class [3] or reordering of the listen queue [12], will have little effect, at least when not used in combination with other schemes. As the experiments in the next section show, using our adaptive architecture the average response time is low. In the experiments without service differentiation, the average response time of the accepted requests is almost always below 150 milliseconds, even when the offered load is very high. Even the 90-percentile of the response time is not high, mostly below 200 milliseconds. This means, that the important task of service differentiation is to make sure that the acceptance rate of premium requests is higher than the acceptance rate of standard requests.

To achieve higher throughput for more important service classes, we divide token buckets into logical partitions or logical buckets [10], one for each service class. Generated tokens are divided between the logical bucket with a specified proportion. When a logical bucket is full, the newly generated tokens for this partition are put into the overflow buffer. Requests can consume these extra tokens when there is no token in their logical partition. This enables requests belonging to one service class to use unused tokens from the other service class.

Figure 2 shows an example with two service classes which are distinguished based on network-level information. TCP SYN policing would use such a scheme. For example, we could choose to reserve two thirds of the tokens for the "premium partition", i.e., for premium clients, and one third for the "standard partition". This should lead to twice the acceptance rate for premium requests compared to standard requests when the offered load is sufficiently high. Requests are matched against filter rules to determine the logical bucket corresponding to their service class. For

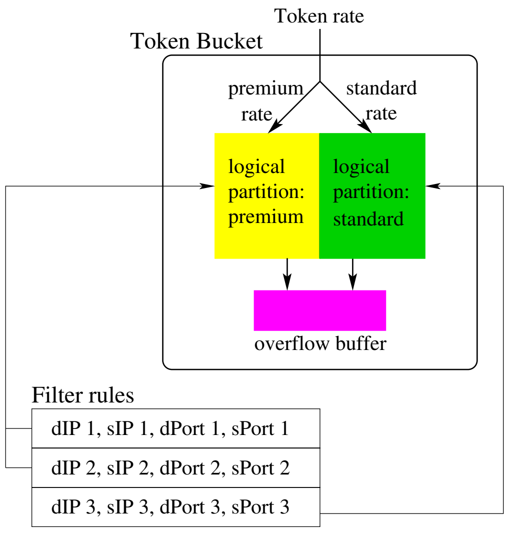

Token rate
Token Bucket
premium rate
standard rate
logical partition: premium
logical partition: standard
overflow buffer
Filter rules
dIP 1, sIP 1, dPort 1, sPort 1
dIP 2, sIP 2, dPort 2, sPort 2
dIP 3, sIP 3, dPort 3, sPort 3

Figure 2. Token bucket with logical partitions

example, the first two filter rules in Figure 2 could contain the IP addresses of premium clients while the last filter rule could be a default rule matching all other requests.

The same scheme can also be used to provide service differentiation for resource-based admission control. For example, the token bucket associated with CPU-intensive requests could be partitioned in the same way as described above.

Note that also the size of the logical buckets plays an important role. Larger bucket sizes lead to higher throughput when the request arrival distribution is bursty. When the bucket size is small, requests need to arrive at the server at regular intervals in order to make full use of the bucket's rate. Therefore, it is meaningful to use a larger bucket size for the premium partition than for the standard partition.

4 Experiments

Our testbed consists of a server and two traffic generators connected via a 100 Mb/sec Ethernet switch. The server machine is a 600 MHz Athlon with 128 MBytes of memory running Linux 2.4. The traffic generators run on a 450 MHz Athlon and a 200 MHz Pentium Pro. The server is an unmodified Apache web server, v.1.3.9.

For client load generation we use the sclient traffic generator [1]. Sclient is able to generate client request rates that exceed the capacity of the web server. This is done by aborting requests that do not establish a connection to the server in a specified amount of time. Sclient in its unmodified version requests a single file. For most of our experiments we have modified sclient to request files according to a workload that is derived from the surge traffic generator [2]. The workload has three important properties: The size of the files stored on the server follows a heavy tailed distribution. The request size distribution is heavy tailed and the distribution of popularity of files follows Zipf's Law. Zipf's Law states that if the files are ordered from most popular to least popular, then the number of references to a file tends to be inversely proportional to its rank.

We separate the files in two directories on the server. The files >50 KBytes are put into one directory (`/islarge`), the smaller files into another directory. We make 20% of the requests for small files dynamic. The dynamic files used in our experiments are minor modifications of standard Webstone [11] CGI files and return a file containing randomly generated characters of the specified size.

For the acceptance rate of both CGI-scripts and large files, minimum rates can be specified. The reason for this is that the processing of CGI-scripts or large files should not be completely prevented even under heavy load. This minimum rate is set to 10 reqs/sec in our experiments.

4.1 CPU Utilization and Listen Queue Length

We use two controllers in this experiment: the CPU controller that adapts the acceptance rate of CGI-scripts and the SYN controller. In the experiment, about 20% of the requests are for CGI scripts. We vary the request rate across runs. The goals of the experiment are the following: First, showing that the control algorithms and in particular resource-based admission control prevent overload. Second, showing that TCP SYN policing becomes active when resource-based admission control alone cannot prevent server overload. Third, demonstrating that the system achieves high throughput and low response times over a broad range of request rates.

When the request rates are low, we expect that no requests should be discarded. When the request rate increases, we expect that the CPU becomes overutilized mostly due to the CPU-intensive CGI-scripts. Hence, for some medium request rates, policing of CGI-scripts is sufficient and TCP SYN policing should not be active. However, when the offered load increases beyond a certain level, the processing capacity of the server will not be able keep up with the request rate, even when discarding most of the CPU-intensive requests. At that point, the listen queue will build up and, therefore, TCP SYN policing will become active.

Figure 3 illustrates the throughput and response times for different request rates. When the request rate is about 375 reqs/sec, the average response time increases and the throughput decreases when no controls are applied. At that point, the CPU becomes overutilized and cannot process requests at the same rate as they arrive. Hence, the listen queue builds up which contributes additionally to the increase of the response time.

Using resource-based admission control, the acceptance rate of CGI-scripts is decreased which prevents the CPU from becoming a bottleneck and hence keeps the response time low. Decreasing the acceptance rate of CGI-scripts is sufficient until the request rate is about 675 reqs/sec. At this point the CGI acceptance rate reaches the predefined minimum and cannot be decreased anymore despite the CPU utilization being greater than the reference value. The reference value is the desired CPU utilization which is set to 90% in our experiments. As the server's processing rate is lower than the request rate, the listen queue starts building up. Due to the increase of the listen queue, the controller computes a lower TCP SYN policing rate which limits the number of accepted requests. This is shown in the left-hand graph where the throughput does not increase anymore for request rates higher than 800 reqs/sec. The right-hand graph shows that the average response time increases slightly when TCP SYN policing is active. This increase is partly caused by the additional waiting time in the listen queue.

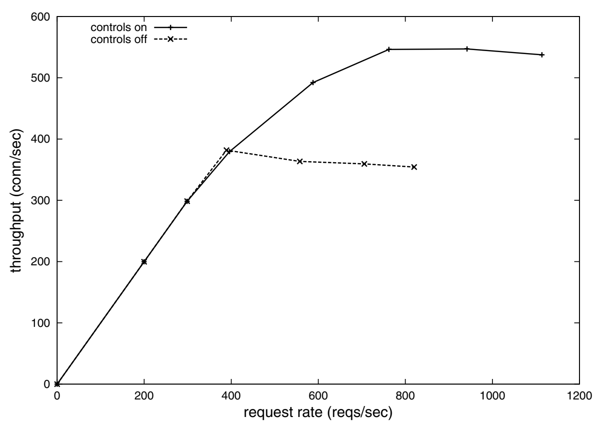
We have repeated this experiment with workloads containing only static requests. If requests are discarded using HTTP header-based control, the onset of TCP SYN policing should happen at higher request rates. Our experiments have shown that this is indeed the case: When the fraction of dynamic requests is 20%, TCP SYN policing sets in at about 675 reqs/sec while the onset for SYN policing is at about 610 reqs/sec when all requests are for static files.

To summarize this experiment, for low request rates, we prevent server overload using resource-based admission control that avoids over-utilization of the resource bottleneck, in this case CPU. For high request rates, when resource-based admission control alone is not sufficient, TCP SYN policing reduces the overall acceptance rate which keeps the response times low and the throughput high.

4.2 Queue Length to the Network Interface

Although the workload used in the previous section contains some very large files, we noticed few packet drops on the network interface. In the experiments in this section we make the bandwidth of the outgoing interface a bottleneck by requesting a large static file of size 142 KBytes from another host. The original host still requests the surge-like workload at a rate of 300 reqs/sec. From Figure 3 we know that the server can cope with the workload from this particular host requested at this rate. The request of the large static file will cause overutilization of the interface and a proportional drop of packets to the original host.

Without admission control, we expect that packet drops on the outgoing interface will cause lower throughput and



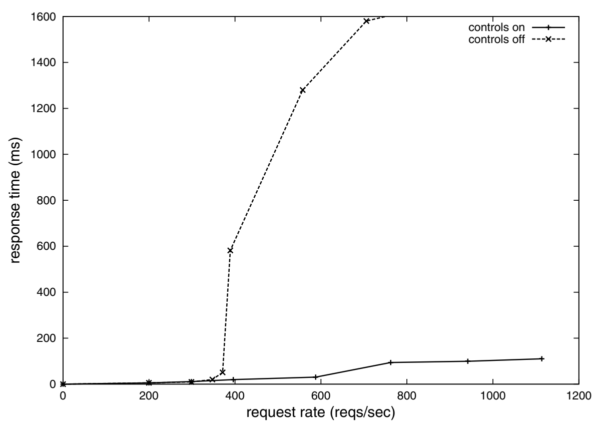


Figure 3. Comparison standard system and system with adaptive overload control

in particular higher average response times by causing TCP to back off due to the dropped packets. Therefore, we insert a rule that controls the rate at which large files are accepted. Large files are identified by a common prefix (`/islarge`). The aim of the experiment is to show that by adapting the rate with that requests for large files are accepted, we can avoid packets drops on the outgoing interface.

Requests to the large file are generated with a rate of 50 and 80 reqs/sec. The results are depicted in Table 1 (page 6). As expected the response times for both workloads become very high when no controls are applied. In our experiments, we observed that the length of the queue to the interface was always around the maximum value which indicates a lot of packet drops. By discarding a fraction of the requests for large files, our controls keep the response time low by avoiding drops in the queue to the network interface. Although the throughput for the large workload is higher when no controls are applied, the sum of the throughput of both workloads is higher using the controls.

4.3 Service Differentiation

The aim of this experiment is to show that our architecture can provide higher throughput to premium clients, using the scheme discussed in Section 3. We have two service classes, premium and standard. We reserve $3/5$ of the tokens generated for the token bucket associated with TCP SYN policing for premium requests, while reserving the remaining tokens for standard requests. The size of the logical bucket for premium requests is 20, while the other logical bucket has a size of five. The size of the overflow buffer is also five. For simplicity, we have fixed the acceptance rate of CGI-scripts to 10 reqs/sec. Sclient requests the surge-workload from two machines. We have two filter rules. One rule matches the IP address of the host emulating premium clients, the other one is a default rule matching the requests from the standard host.

service class	throughput (reqs/sec)
premium	331.1
standard	222.1
sum	553.2

Table 2. Throughput for each service class (equal request rates)

service class	throughput (reqs/sec)
premium	429.5
standard	120.9
sum	550.4

Table 3. Premium class using unused tokens from standard class

In the first experiment we request the same workload from both hosts at a rate of 450 reqs/sec each. We expect that TCP SYN policing is active at this request rate. Of the accepted requests about 60% should be premium and 40% should be standard requests. The results are shown in Table 2. The overall throughput is 553.2 reqs/sec. The throughput of the premium requests is 331.1 reqs/sec which is about 60% of the total throughput.

In the next experiment we want to show that the unused tokens of one service class can be utilized by the other service class using the overflow buffer. The request rate for premium requests is 760 reqs/sec while the request rate for standard requests is about 150 reqs/sec. Again, the acceptance rate of CGI-scripts is fixed to 10 reqs/sec. The token rate for the standard partition of the token bucket will be higher than 150 tokens per second. Thus, this logical bucket will become full and the exceeding tokens are put into the overflow buffer where they can be consumed

req rate large workload	metric	large workload		surge workload	
		no controls	controls	no controls	controls
50 reqs/s	tput (reqs/sec)	46.8	41.5	270.7	289.2
50 reqs/s	response time (ms)	2144	80.5	1394.8	26.9
80 reqs/s	tput (reqs/sec)	55.5	45.8	205.2	285.1
80 reqs/s	response time (ms)	5400	94	3454.5	29.3

Table 1. Outgoing bandwidth

by premium requests. The results are shown in Table 3. The throughput for standard requests is about 121 reqs/sec, i.e. the requested rate minus most of the dynamic requests. The throughput for premium requests is 429.5 requests/sec, i.e. more than 60% of the overall acceptance rate. TCP SYN policing accepts slightly below 700 reqs/sec. However, most of the dynamic requests are then discarded by the resource-based admission control. These results demonstrate that the service differentiation mechanism works as expected.

5 Discussion

The proposed solution of grouping the objects according to resource demand in the web server's directory tree, is not intuitive and awkward for the system administrator. We assume that this process can be automated using scripts.

Since our basic architecture is implemented as a kernel module, we have decided to put the control loops in the kernel module as well. An advantage of having the control mechanisms in the kernel is that they are actually executed at the correct sampling rate. The same mechanisms could be deployed in user space.

Our kernel module is not part of the TCP/IP stack which makes it easy to port the mechanisms. The only requirements are availability of timing facilities to ensure correct sampling rates and facilities to monitor resource utilization.

6 Conclusions

We have presented an adaptive server overload protection architecture for web servers. The architecture uses resource-based admission control to avoid overutilization of critical web server resources. The architecture also provides service differentiation using token buckets with logical partitions. Our experiments have shown that the acceptance rates are adapted as expected. Our system sustains high throughput, in particular for premium requests, and low response times even under high load.

Acknowledgements

This work builds on the architecture that has been developed at IBM TJ Watson together with R. Tewari, A. Mehra and D. Freimuth [12]. Without them, this work would not have been possible. Thanks to M. Sanfridson and J. Carlström for discussions on the control algorithms and A. Bavier, I. Marsh, A. Pears and B. Ahlgren for comments on earlier drafts of this paper.

References

[1] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, Dec. 1997.

[2] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of SIGMETRICS*, 1998.

[3] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, Sept. 1999.

[4] P. Bhoj, S. Ramanathan, and S. Singhal. Web2k: Bringing QoS to web servers. Technical report, HP, May 2000.

[5] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical report, HP, 1999.

[6] S. Parekh *et al.* Using control theory to achieve service level objectives in performance management. In *International Symposium on Integrated Network Management*, May 2001.

[7] T. Glad and L. Ljung. *Reglerteknik: Grundläggande teori (in Swedish)*. Studentlitteratur, 1989.

[8] H. Jamjoom and J. Reumann. Qguard: Protecting internet servers from overload. Technical report, University of Michigan, 2000.

[9] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Real-Time Technology and Application Symposium*, June 2001.

[10] A. Mehra, R. Tewari, and D. Kandlur. Design considerations for rate control of aggregated TCP connections. In *Proc. of NOSSDAV*, June 1999.

[11] Mindcraft. Webstone. http://www.mindcraft.com.

[12] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Usenix Annual Technical Conference*, June 2001.