Resource-based Web Server Overload Protection

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

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

Abstract

The main web server resources are the network interface, CPU and disk. Depending on the workload, some server resources can be overutilized while the demand on other resources is low because certain types of requests utilize one resource more than others. In this paper, we present an architecture that performs admission control based on the current server resource utilization combined with knowledge about resource consumption of requests. Our experiments demonstrate more than 40% higher throughput during overload compared to a standard server and several magnitudes lower response times.

1 Introduction

The increasing number of Internet users and innovative new services such as e-commerce are placing new demands on web servers. It is becoming essential for web servers to be highly available, have fast response times, and provide continuous service during overload.

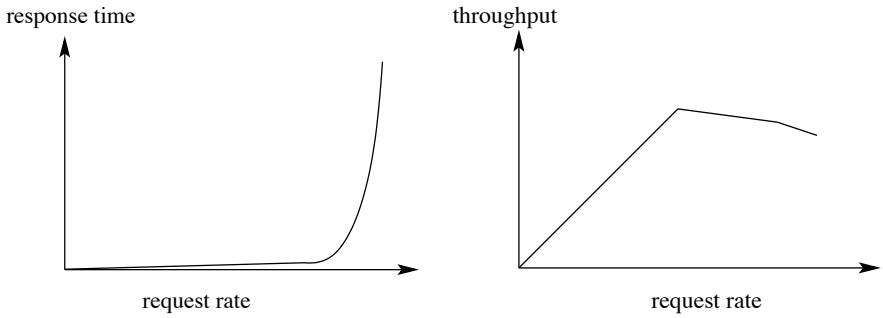

Figure 1: Impact of server overload on response time and throughput

Web servers become overloaded when one or several server resources become overutilized. As shown in Figure 1, server overload affects both the server throughput and the response time experienced by the clients. The left part of Figure 1 demonstrates how the response time increases with the server load and the right part depicts the decrease in server throughput. The response time is low as long as no server resource is overutilized. However, when the server resource bottleneck becomes overutilized, i.e., the service rate of the bottleneck cannot keep up with the arrival rate of jobs/requests, the queue length to the resource bottleneck and thus the response time theoretically increases to infinity. This is depicted by the sudden increase of the response time. One of our goals is to avoid this increase of the response time during overload.

The server throughput increases with the request rate until the request rate exceeds the capacity of the web server. At this point, the throughput decreases due to the additional time the CPU spends on processing

*Thiemo is also at Uppsala University. This work is partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

incoming connection requests that are dropped when the listen queue is full. Lower server throughput leads to loss of revenue, while long delays cause user frustration and decrease task success and efficiency.

The main server resources are the network interface, CPU and disk [10]. Any of these may become the server's bottleneck, depending on the kind of workload the server is experiencing [12]. For example, the majority of CPU load is caused by a few CGI requests [4]. The network interface typically becomes overutilized when the server concurrently transmits several large files.

While most of the proposed architectures for server overload protection throttle requests when one resource is overutilized, the main contribution of this work is an architecture that avoids server overload by preventing overutilization of specific server resources. This is achieved using adaptive inbound controls. We utilize the information found in the HTTP header of incoming requests with knowledge about the resource consumption of requests to avoid resource overutilization and server overload. We call our approach resource-based admission control.

Resource-based admission control deploys a kernel-based mechanism for overload protection and service differentiation called *HTTP header-based connection control* [22]. HTTP header-based connection control performs admission control based on application-level information such as URL, sets of URLs (identified by, for example, a common prefix), type of request (static or dynamic) and cookies. HTTP header-based control deploys token bucket policers for admission control. HTTP header-based connection control is used in conjunction with filter rules that specify application-level attributes and the parameters for the associated control mechanism, i.e. the rate and bucket size of the policer.

When the request rate reaches above a certain level, resource-based admission control alone cannot prevent overload, for example during flash crowds or Denial-of-Service (DoS) attacks. When such situations arise, we use *TCP SYN policing* [22]. This mechanism is efficient in terms of resource consumption of rejected requests because it provides "early discard". The admission of connection requests is based on network-level attributes such as IP addresses and a token bucket policer.

Another contribution of this work are mechanisms that dynamically set the rate of the token bucket policers based on the utilization of the critical resources. Since the web server workload frequently changes, for example when the popularity of documents or services changes, assigning static rates that work under these changing conditions may either lead to underutilization of the system when the rates are too low or there is a risk for overload when the rates are too high. The adaptation of the rates is done using feedback control loops. Techniques from control theory have been used successfully in server systems before [2, 17, 13, 9].

We have implemented this admission control architecture in the Linux OS and conducted experiments in a controlled network using an unmodified Apache web server. Experiments demonstrate that overload protection and adaptation of the rates works as expected. Our results show that our architecture keeps the response times lower and the throughput higher during overload compared to a standard Apache on Linux configuration.

The remainder of the paper is structured as follows: In the next section we present related work. Section 3 presents the system architecture including the controllers. Section 4 presents experiments that evaluate various aspects of our system. Section 5 discusses architectural extensions and further insights. Finally, Section 6 concludes the paper.

2 Related Work

Several research efforts have focused on overload control and service differentiation in web servers [3, 8, 15, 11]. *WebQoS* [8] is a middleware layer that provides service differentiation and admission control. Since it is deployed in middleware, it is less efficient compared to kernel-based mechanisms. Cherkasova et al. [11] present an enhanced web server that provides session-based admission control to ensure that longer sessions are completed. Their scheme is not adaptive and rejects new requests when the CPU utilization of the server exceeds a certain threshold. The focus of cluster reserves [5] is to provide performance isolation in cluster-based web servers by managing resources, in their work CPU. Their resource management and distribution strategies do not consider multiple resources.

Also other have used approaches from control theory for server systems. Abdelzaher and Lu [2] use a control loop to avoid server overload and meet individual deadlines for all served requests. They express

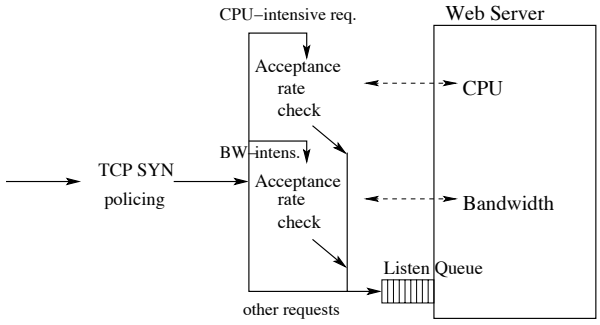

Figure 2: Admission control architecture

server utilization as a function of the served rate and the delivered bandwidth [1]. Their control task is to keep the server utilization at $ln2$ in order to guarantee that all deadlines can be met. In our approach we aim for higher utilization and throughput. Furthermore, our approach also handles dynamic requests. In another paper, Lu et al. [17] use a feedback control approach for guaranteeing relative delays in web servers. In their work, large delays are caused by HTTP 1.1 that keeps persistent connections open to await further requests in combination with the small number of concurrent connections Apache servers can handle. Parekh et al. [13] use a control-theoretic approach to regulate the maximum number of users accessing a Lotus Notes server. While a focus of these papers is to use control theory to avoid the absence of oscillations, Bhoj et al. [9] in a similar way as we, use a simple controller to ensure that the occupancy of the priority queue of a web server stays at or below a pre-specified target value. Reumann et al. [16] use a mechanism similar to TCP SYN policing to avoid server overload.

3 Architecture

The Admission Control Architecture

Figure 2 illustrates our admission control architecture. In the right part of the figure we see the web server and some of its critical resources. We use *HTTP header-based connection control* to avoid overutilization of critical resources. HTTP header-based connection control is activated when the HTTP header is received. Using this mechanism an informed control is possible which provides the ability to, for example, specify lower access rates for CGI requests than other requests that are less CPU-intensive. This is done using filter rules, e.g. checking URL, name and type. With each of the critical resources, a filter rule and a token bucket policer is associated. 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. For example, a filter rule `/cgi-bin` and an associated token bucket policer restrict the acceptance of CPU-intensive requests. On receipt of a request, the HTTP header is parsed and matched 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. We call this part of our admission control architecture resource-based admission control.

For each of the critical resources, we use a feedback control loop that adapts the token rate at which we accept requests in order to avoid overutilization of the request. Note that 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. TCP SYN policing limits acceptance of new SYN packets based on compliance with a token bucket policer. TCP SYN policing enables service differentiation based on information in the TCP and IP headers of incoming connection requests (i.e, the source and destination addresses and port numbers). 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. Also for TCP SYN policing,

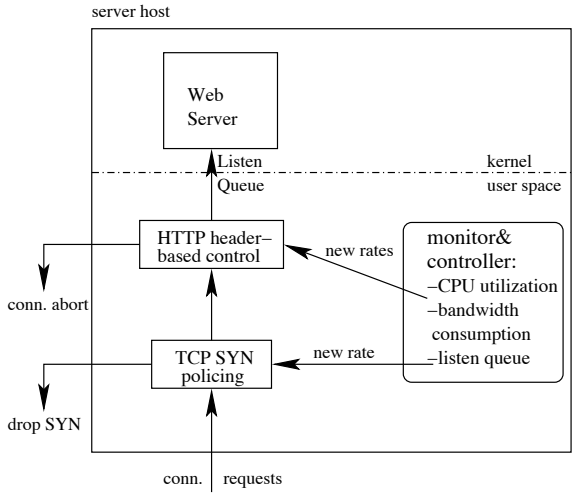

Figure 3: The control architecture

we adapt the token rate while keeping the bucket size fixed.

Both HTTP header-based connections control and TCP SYN policing are located in the kernel which avoids the 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 [22]. 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.

One of our major design goals for the adaptation mechanisms is to keep TCP SYN policing inactive while resource-based admission control can protect resources from being overutilized. Resource-based admission control cannot only check for URLs but also for other application-level information, such as cookies. This gives us the ability to identify ongoing sessions or premium customers while SYN policing cannot access such information.

The Control Architecture

Figure 3 shows the control architecture. The monitor measures the utilization of each critical resource and passes the values to the controller. Based on these values, the controllers adapt the rates for the admission control mechanisms. We deploy one controller for the CPU utilization and one for the bandwidth on the outgoing network interface. We call the former CPU controller and the latter bandwidth controller. 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. 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, called SYN controller, controls the rate of the TCP SYN policer.

Adaptation of the Rates

Different resources have different properties. Therefore, we do not use the same type of controller for each resource. Doing some initial experiments, we found that the simplest resource to control is the CPU since the CPU utilization changes directly with the rate of CPU-intensive requests the server is exposed to. This allows us to use a simple controller, namely a proportional (P) controller. The equation that computes the new rates is called the *control law*. The control law for our P-controller is:

$$rate_{\text{cgi}}(t+1) = rate_{\text{cgi}}(t) + K_{\text{P\_CPU}} * e(t) \quad (1)$$

where $e(t) = CPU\_util_{\text{ref}} - CPU\_util(t)$, i.e. the difference between the *reference* or desired CPU utilization and the current, measured CPU utilization. $rate_{\text{cgi}}(t)$ is the acceptance rate for CGI-scripts at time t. $K_{\text{P\_CPU}}$ is called the *proportional gain*. It determines how fast the system will adapt to changes in the workload. For larger $K_{\text{P\_CPU}}$ the adaptation is faster but the system is less stable and may experience oscillations [14].

The other two controllers base their control laws on the length of queues: The bandwidth controller on the length of the queue to the network interface and the SYN controller on the length of the listen queue. The significant aspect here is actually the change in the queue length since this derivative reacts faster than a proportional factor. The fast reaction is more crucial for these controllers, because the delay between the acceptance decision and the actual occurrence of high resource utilization is higher than when controlling CPU utilization. For example, the delay between accepting too many large requests and overflow of the queue to the network interface is non-negligible. One reason for this is that it takes several round-trip times until the TCP congestion window is sufficiently large to contribute to overflow of the queue to the network interface.

Therefore, we decided to use a proportional derivative (PD) controller for these two controllers. The derivative is approximated by the difference between the current queue length and the previous one, divided by the number of samples. The control law for our PD-controllers is:

$$rate(t+1) = rate(t) + K_{\text{P\_Q}} * e(t) + K_{\text{D\_Q}} * (qlen(t) - qlen(t-1)) \tag{2}$$

where $e(t) = qlen_{\text{ref}} - qlen(t)$. The division is embedded in $K_{\text{D\_Q}}$. $K_{\text{D\_Q}}$ is the *derivative gain*.

Since naive application of Equation 1 results in an increase in the acceptance rate when the measured value is below the reference value, i.e. the resource could be utilized more, we imposed some conditions on the equations above. It is not meaningful to increase the acceptance rate, when the filter rule has less hits than the specified token rate. For example, if we allow 50 CGI requests/sec, the current CPU utilization is 60%, the reference value for CPU utilization is 90% and the server has received 30 CGI requests during the last second, it does not make sense to increase the rate to more than 50 CGI requests/sec. On the contrary, if we increase the rate in such a situation, we would end up with a very high acceptance rate after a period of low server load. Hence, when the measured CPU utilization is lower than the reference value, we have decided to update the acceptance rate only when the number of hits was at least 90% of the acceptance rate during the previous sampling period.

A similar condition is imposed on the SYN and the bandwidth controller. Since both the listen queue and the queue to the network interface very often have a length of zero, their measured length is below the reference value. For the same reason as in the discussion above, if the queue length is below the reference value, we update the acceptance rate only when the length of the queue has changed.

When performing resource-based admission control, we do not police all requests, even if all requests consume resources at least some CPU. Hence, if the CPU utilization is already high, i.e. it is above the reference value, we do not want to increase the amount of work that enters the system since this might cause server overload. Thus, we increase the TCP SYN policing rate only when the CPU utilization is below the reference value.

Another important parameter is the sampling rate. For ease of implementation, we started with a sampling rate of one second. Since even slow web servers can process several hundred requests per second, a sampling rate of one second might be considered long. The question is if we do not miss important events such as the listen queue filling up. This would be the case given that the requests entering the server during one second was not limited. However, TCP SYN policing limits the number of requests entering the system. This bounds the system state changes between sampling points and allows us to use a sampling rate of one second.

4 Experimental Evaluation

The testbed for our experiments comprises a server and two traffic generators connected via a 100 Mb/sec Ethernet switch. The server host 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., with the default configuration.

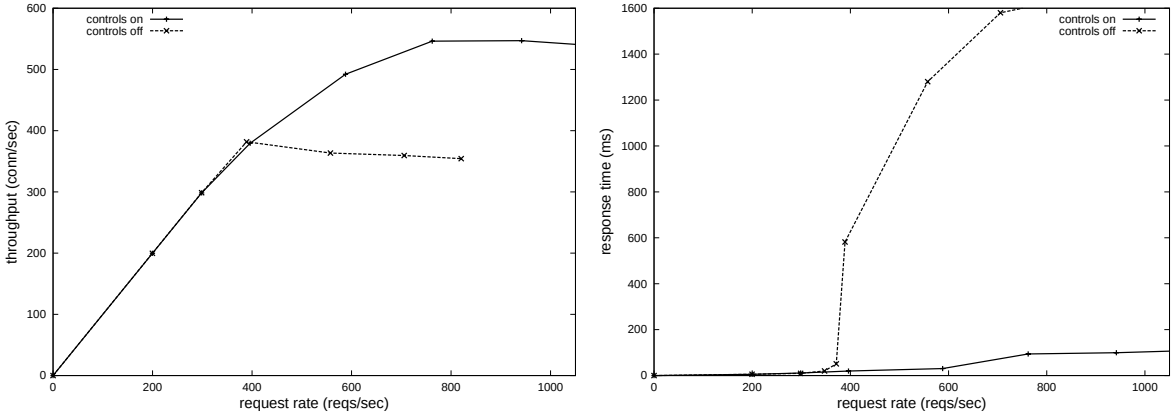

Figure 4: Comparison standard system and system with adaptive overload control

For the control algorithms we have used the following values: The reference values for the queues are set to 100 for the listen queue and 35 for the queue to the network interface. The latter has a length of 100. These values are chosen arbitrarily but they can be chosen lower without significant impact on the stability since the queue lengths are mostly zero. The reference value for CPU utilization is 90%. We chose this value since it allows us to be quite close to the maximum utilization while higher values would more often lead to 100% CPU utilization during one sampling period. The proportional gain for the CPU-controller is set to 1/5. The proportional gain for the SYN and bandwidth controller is set to 1/16, the derivative gain to 1/4. These values were obtained by experimentation. We consider these values to be specific to the server machine we are using. However, we expect them to hold for all kinds of web workloads for this server since we use a realistic workload as described in the next section. The bucket size of the token bucket used for TCP SYN policing is set to 20 unless explicitly mentioned. The token buckets for HTTP header-based controls have a bucket size of five.

Workload

For workload generation we use the sclient traffic generator [6]. 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 [7]. 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.

Determining the total number of requests for each file on the server is also done using surge. The dynamic files used in our experiments are minor modifications of standard Webstone [19] CGI-scripts. We separated the static files in two directories on the server. The files larger than 50 KBytes were put into one directory (`/islarge`), the smaller files into another directory. 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 Supervising the CPU Utilization and the 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 dynamic files. 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

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

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 will 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 4 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. Since our workload contains CPU-intensive CGI-scripts, the CPU becomes overutilized and cannot process requests with 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. 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.

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 is not sufficient, TCP SYN policing reduces the overall acceptance rate which keeps the response times low and the throughput high.

4.2 Supervising the 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 4 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 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. 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

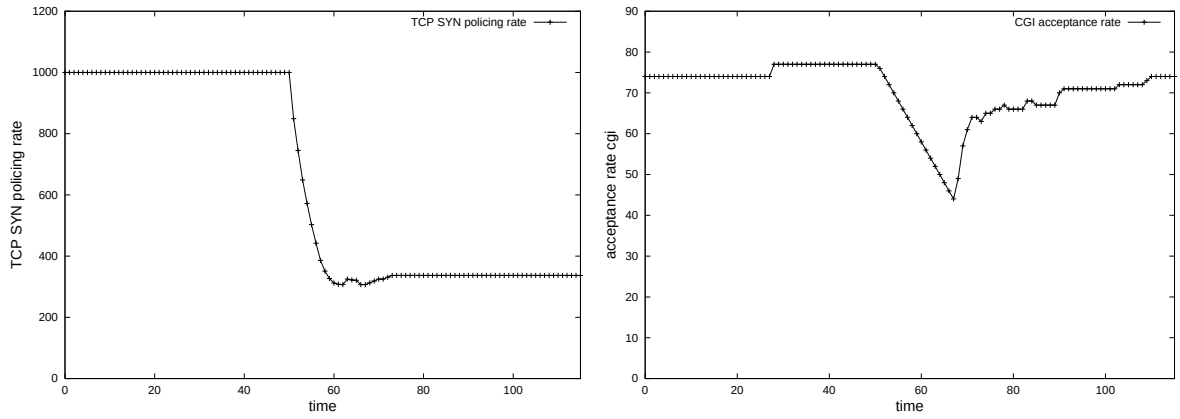

Figure 5: Adaptation under high load (left SYN policing rate, right CGI acceptance rate)

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 for both workloads is higher using the controls. When controls are applied the sum of the throughput for both workloads is the same for both request rates.

4.3 Sudden High Load

The objective of this experiment is to expose the server to a sudden high load and study the behaviour of the control algorithms. Such a load exposure could occur during a flash crowd or a DoS attack. We start with a relatively low request rate of 300 reqs/sec. After 50 seconds we increase the offered load to 850 reqs/sec and sustain this high request rate for 20 seconds before we decrease it to 300 reqs/sec again. We set the initial TCP SYN policing rate to 1000 reqs/sec.

From Figure 5, we see that the TCP SYN policing rate decreases very quickly when the request rate is increased at time 50. This rapid decrease is caused by both parts of the control algorithms in Equation 2. Since the length of the listen queue increases quickly, the contribution of the derivative part is high. Since the absolute length of the listen queue at that time is higher than the reference value, the contribution of the proportional part of Equation 2 is high as well. The TCP SYN policing rate does not increase to 1000 again after the period of high load. However, we can see that around time 70, the policing rate increases to around 340 which is sufficiently high so that no requests need to be discarded by the SYN policer when the request rate is 300 reqs/sec. The SYN policing rate settles at higher rates for higher request rates after the period of high load.

As expected, the CGI acceptance rate does not decrease as fast. Since $K_{\text{P\_CPU}}$ has a value of 1/5, the CGI acceptance rate is decreased at most with two per sampling point during the period of high load. Figure 5 also shows that the CGI-acceptance rate is restored fast after the period of high load. At a request rate of 300 reqs/sec, the CPU utilization is between 70 and 80%. Thus, the absolute difference to the reference value is larger than during the period of high load which enables faster increase than decrease of the CGI acceptance rate. At time 30 in the right-hand graph, we can see the CGI acceptance rate jump from 74 to 77. The reason for this jump is that during the last sampling period, the number of hits for the corresponding filter rule was above 90%, while it was otherwise below 90% until time 50.

5 Discussion

The sclient program generates web server requests at a constant rate. The resulting requests also arrive at a constant rate to the web server. We have modified the sclient program to generate requests following a Poisson distribution with a given mean. In another experiment, we have shown that our adaptation mechanisms should be able to cope with bursty traffic provided we make a sensible choice of the bucket size [21].

Although our architecture is implemented as a kernel module, it could also be deployed in user space or in a middleware layer. 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 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.

The interaction between the different control loops might cause oscillations. Fortunately, requests to large static files do not consume much CPU while CPU-intensive requests usually do not consume much network bandwidth. Hence, we assume that the control loops for these resources will not experience any significant interaction effects. The adaptation the TCP SYN policing rate affects the number of accepted CPU-intensive and bandwidth-intensive requests, which may cause interactions between the control loops. By increasing rates quite conservatively we hope to avoid this effect. Furthermore, in none of our experiments we have seen an indication that such an interaction might actually occur. The main reason for this is that TCP SYN policing becomes active when the acceptance rate of CPU-intensive requests is very low and most of the CPU-intensive requests are discarded.

Extensions of the Architecture

In our adaptive architecture accepted requests can be processed quickly since server resources are not overutilized and the listen queue is short. This leads to low response times. Therefore, the major goal of service differentiation in our architecture is to provide high throughput to premium customers. This can be done by splitting the token buckets used for admission control into logical partitions [18]. Each logical partition corresponds to one service class with larger partitions for more important service classes.

Our current implementation is targeted towards single node servers or the back-end servers in a web server cluster. We believe that the architecture can easily be extended to LAN-based web server clusters and enhance sophisticated request distribution schemes such as Harvard Array of Clustered Computers HACC [23] and Locality-aware Request Distribution (LARD) [20]. In both LARD and HACC, the front-end distributes requests based on locality of reference to improve cache hit rates and thus increase performance. In the extended architecture, the front-end performs resource-based admission control. The back-end servers monitor the utilization of each critical resource and propagate the values to the front-end. Based on these values, the front-end updates the rates for the token bucket based policers using the algorithms. After the original distribution scheme has selected the node that is to handle the request, compliance with the corresponding token bucket ensures that critical resources on the back-ends are not overutilized. This way, we consider the utilization of individual resources as a distribution criteria which neither HACC nor LARD do. HACC explicitly combines these performance metrics into a single load indicator. There are two potential problems: First, the need to propagate the values from the back-ends to the front-end causes some additional delay. If the evaluation of the system shows that this is indeed a problem, we should be able to overcome it by setting more conservative reference values or by increasing the sampling rate. Second, there is a potential scalability problem caused by the need for $n * c$ token buckets on the front-end, where n is the number of back-ends and c the number of critical resources. However, we believe that this is not a significant problem since a token bucket can be implemented by reading the clock (which in kernel space is equal to reading a global variable) and performing some arithmetical operations.

6 Conclusions

In this paper, we have presented an adaptive server overload protection architecture for web servers. Using the application-level information in the HTTP header of the requests combined with knowledge about resource consumption of resource-intensive requests, the system adapts the rates at which requests are accepted. The architecture combines the use of such resource-based admission control with TCP SYN policing. TCP SYN policing first comes into play when the load on the server is very high since it wastes less resources when rejecting requests. Our experiments have shown that the acceptance rates are adapted as expected. Our system sustains high throughput and low response times even during overload.

Acknowledgements

This work builds on the architecture that has been developed at IBM TJ Watson together with Renu Tewari, Ashish Mehra and Douglas Freimuth [22]. Without them, this work would not have been possible. Thanks to Andy Bavier, Ian Marsh, Arnold Pears and Bengt Ahlgren for valuable comments on earlier drafts of this paper. The authors also want to thank Martin Sanfridson and Jakob Carlström for discussions on the control algorithms.

References

[1] T. Abdelzaher and N. Bhatti. Web server qos management by adaptive content delivery. In *Int. Workshop on Quality of Service*, June 1999.

[2] T. Abdelzaher and C. Lu. Modeling and performance control of internet servers. In *IEEE Conference on Decision and Control*, December 2000.

[3] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Proc. of Internet Server Performance Workshop*, March 1999.

[4] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM Sigmetrics*, April 1996.

[5] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proc. of ACM SIGMETRICS*, June 2000.

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

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

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

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

[10] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for web clusters providing multiple services. In *Proc. of 10th Int'l World Wide Web Conference*, May 2001.

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

[12] L. Eggert and J. Heidemann. Application-level differentiated services for web servers. *World Wide Web Journal*, 3(2):133–142, September 1999.

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

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

[15] R. Iyer, V. Tewari, and K. Kant. Overload control mechansims for web servers. In *Performance and QoS of Next Generation Networks*, November 2000.

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

[17] 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.

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

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

[20] V. Pai, M. Aron, G. Banga, M. Svendsen, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, USA, October 1998.

[21] T. Voigt and P. Gunningberg. Handling multiple bottlenecks in web servers using adaptive inbound controls. In *Int. Workshop on Protocols for High-Speed Networks*, Berlin, Germany, April 2002.

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

[23] X. Zhang, M. Barrientos, J. Chen, and M. Seltzer. HACC: An architecture for cluster-based web servers. In *Third Usenix Windows NT Symposium*, pages 155–164, Seattle, WA, July 1999.