Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls

Thiemo Voigt[1] and Per Gunningberg[2]

[1] SICS, Box 1263, SE-16429 Kista, Sweden thiemo@sics.se
[2] Uppsala University, Box 337, SE-75105 Uppsala, Sweden
Per.Gunningberg@it.uu.se

Abstract. Web servers become overloaded when one or several server resources are overutilized. In this paper we present an adaptive architecture that prevents resource overutilization in web servers by performing admission control based on application-level information found in HTTP headers and knowledge about resource consumption of requests. In addition, we use an efficient early discard mechanism that consumes only a small amount of resources when rejecting requests. This mechanism first comes into play when the request rate is very high in order to avoid making uninformed request rejections that might abort ongoing sessions. We present our dual admission control architecture and various experiments that show that it can sustain high throughput and low response times even during high load.

1 Introduction

Web servers need to be protected from overload because web server overload can lead to high response times, low throughput and potentially loss of service. Therefore, there is a need for efficient admission control to maintain high throughput and low response time during periods of peak server load. Servers become overloaded when one or several critical resources are overutilized and become the bottleneck of the server system. The main server resources are the network interface, CPU and disk [8]. Any of these may become the server's bottleneck, depending on the kind of workload the server is experiencing [10]. 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.

In this paper, we report on an adaptive admission control architecture that utilizes the information found in the HTTP header of incoming requests. Combining this information with knowledge about the resource consumption we can avoid resource overutilization and server overload. We call our approach resource-based admission control.

The current version of our admission control architecture is targeted at overloaded single node web servers or the back-end servers in a web server cluster. The load on web servers can be reduced by distributed web server architectures that distribute client requests among multiple geographically dispersed server nodes in a user-transparent way [29]. Another approach is to redirect requests to web caches. For example, in the distributed cache system Cachemesh [30] each cache server becomes the designated cache for several web sites. Requests for objects not in the cache are forwarded to the responsible cache. However, not all web data is cacheable, in particular dynamic and personalized data. Also, load balancing mechanisms on the front-ends of web server clusters can help to avoid overload situations on the back-end servers. However, sophisticated load-balancing cannot replace proper overload control [13]. Another existing solution to alleviate the load on web servers based on a multi-process architecture such as Apache is to limit the maximum number of server processes. However, this approach limits the number of requests that the server can handle concurrently and can lead to performance degradation.

The main contribution of this work is an adaptive admission control architecture that handles multiple bottlenecks in server systems. Furthermore, we show how we can use TCP SYN policing and HTTP header-based control in a combined way to perform efficient and yet informed web server admission control.

Resource-based admission control uses a kernel-based mechanism for overload protection and service differentiation called *HTTP header-based connection control* [31]. HTTP header-based connection control allows us to perform 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 uses 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.

Our idea to avoid overutilization of server resources is the following: we collect all objects that when requested are the main consumers of the same server resource into one directory. Thus, we have one directory for each critical resource. Each of these directories is then moved into a separate part of the web server's directory tree. We associate a filter rule with each of these directories. Hence, we can use HTTP header-based control to protect each of the critical resources from becoming overutilized. For example, CPU-intensive requests reside in the `cgi-bin` directory and a filter rule specifying the application-level information (URL prefix `/cgi-bin`) is associated with the content at this location.

When the request rate reaches above a certain level, resource-based admission control cannot prevent overload, for example during flash crowds. When such situations arise, we use *TCP SYN policing* [31]. This mechanism is efficient in terms of resource consumption of rejected requests because is provides "early discard". The admission of connection requests is based on network-level attributes such as IP addresses and a token bucket policer.

This paper also presents novel 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, 14, 11, 20].

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

The rest of the paper is structured as follows. Section 2 presents the system architecture including the controllers. Section 3 presents experiments that evaluate various aspects of our system. Section 4 discusses architectural extensions. Before concluding, we present related work in Section 5.

2 Architecture

2.1 Basic Architecture

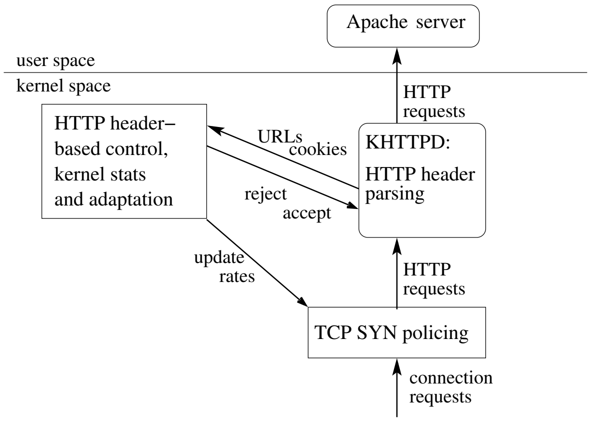

Fig. 1. Kernel-based architecture

Our basic architecture deploys mechanisms for overload protection and service differentiation in web servers that have been presented earlier [31]. These mechanisms control the amount and rate of work entering the system (see Figure 1).

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 incoming connection request (i.e, the 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 CGI requests than other requests that are less CPU-intensive. This is done using filter rules, e.g. checking URL, name and type.

The architecture consists of an unmodified Apache server, the TCP SYN policer, the in-kernel HTTP GET engine khttpd [24] and a control module that implements HTTP header-based control, monitors critical resources and adapts the acceptance rates. Khttpd is used for header parsing only. After parsing the request header it passes the URLs and cookies to the control module that performs HTTP header-based control. If the request is rejected, khttpd resets the connection. In our current implementation, this is done by sending a TCP RST back to the client. If the request is accepted it is added to the Apache web server's listen queue. TCP SYN policing drops non-compliant TCP SYNs. This implies that the client will time-out waiting for the SYN ACK and retry with an exponentially increasing time-out value. For a more detailed discussion, see [31].

Both mechanisms are located in the kernel which avoids the context switch to user space for rejected requests. The kernel mechanisms have proven to be much more efficient and scalable than the same mechanisms implemented in the web server [31].

2.2 The Dual Admission Control Architecture

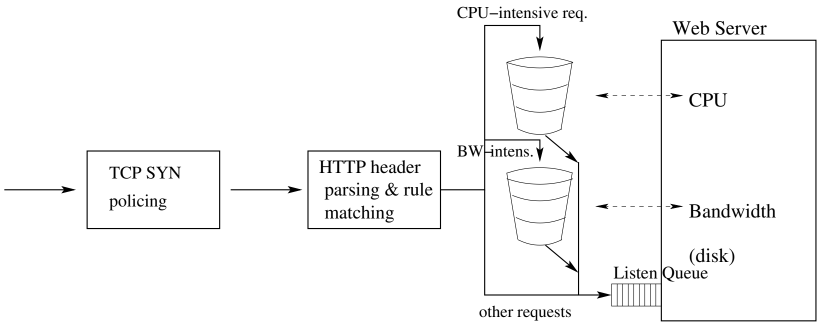

Fig. 2. Admission control architecture

Our dual admission control architecture is depicted in Figure 2. In the right part of the figure we see the web server and some of its critical resources. With each of these resources, a filter rule and a token bucket policer is associated to avoid overutilization of the resource, i.e. we use the HTTP header-based connection control mechanism. 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 to adapt the token rate at which we accept requests in order to avoid overutilization of the request. We do not adapt the bucket size but assume it to be fixed. Note, that the choice of the policer's bucket size is a trade-off [31]. When we have a large bucket size, malicious clients can send a burst to put high load on the machine, whereas when the bucket size is small, clients must come at regular intervals to make full use of the token rate. Furthermore, when the bucket size is smaller than the number of parallel connections in a HTTP 1.0 browser, a client might not be able to retrieve all the embedded objects in a HTML-page since requests for these objects usually come in a burst after the client has received the initial HTML page.

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. We could 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 this is TCP SYN policing's early discard capability. Also for TCP SYN policing, we adapt the token rate and keep the bucket size fixed.

One of our design goals for the adaptation mechanisms is to keep TCP SYN policing inactive while resource-based admission control can protect resources from being overutilized. When performing resource-based admission control, the whole HTTP header has been received and can be checked not only for URLs but also for other application-level information, such as cookies. This gives us the ability to identify ongoing sessions or premium customers. TCP SYN policing's admission control is based on network-level information only and cannot assess such application-level information. Note that this does not mean that TCP SYN policing is not worthwhile [31]. Firstly, TCP SYN policing can provide service differentiation based on network-level attributes. Secondly, TCP SYN policing is more efficient than HTTP header-based control in the sense that less resources are consumed when a request is discarded.

Our architecture uses several control loops to adapt the rate of the token bucket policers: One for each critical resource and one to adapt the SYN policing rate. A consequence of this approach is that 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. Thus, we believe that the control loops for these resources will not experience any significant interaction effects. Adapting the TCP SYN policing rate affects the number of CPU-intensive and bandwidth-intensive requests, which may cause interactions between the control loops. To avoid this effect, we increase rates quite conservatively. Furthermore, in none of our experiments we have seen an indication that such an interaction might actually occur. One of the reasons for this is that TCP SYN policing becomes active when the acceptance rate of CPU-intensive requests is very low, i.e. most of the CPU-intensive requests are discarded.

2.3 The Control Architecture

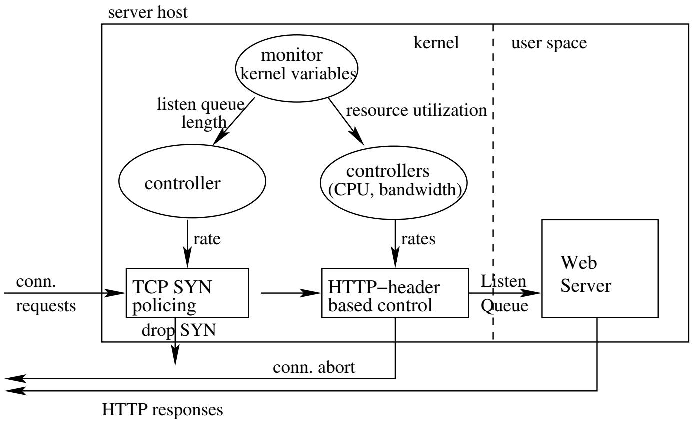

Fig. 3. The control architecture

Our control architecture is depicted in Figure 3. The monitor's task is to monitor the utilization of each critical resource and pass it to the controller. The controllers adapt the rates for the admission control mechanisms. We use 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.

Since different resources have different properties, we cannot use the same controller for each resource. The simplest resource to control is the CPU. The CPU utilization changes directly with the rate of CPU-intensive requests. This makes it possible to use a proportional (P) controller. The equation that computes the new rates is called the *control law*. For our P-controller the control law is:

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

where $e(t) = CPU\_util_{\mathrm{ref}} - CPU\_util(t)$, i.e. the difference between the *reference* or desired CPU utilization and the current, measured CPU utilization. $rate_{\mathrm{cgi}}(t)$ is the acceptance rate for CGI-scripts at time t. $K_{\mathrm{P\_CPU}}$ is called the *proportional gain*. It determines how fast the system will adapt to changes in the workload. For higher $K_{\mathrm{P\_CPU}}$ the adaptation is faster but the system is less stable and may experience oscillations [12].

The other two controllers in our architecture base their control laws on the length of queues: The SYN controller on the length of the listen queue and the bandwidth controller on the length of the queue to the network interface. The significant aspect here is actually the change in the queue length. This derivative reacts faster than a proportional factor. This fast reaction is more crucial for these controllers since 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.

We decided therefore 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:

$$rate(t+1) = rate(t) + K_{\mathrm{P\_Q}} * e(t) + K_{\mathrm{D\_Q}} * (queue\_len(t) - queue\_len(t-1)) \quad (2)$$

where $e(t) = queue\_len_{\mathrm{ref}} - queue\_len(t)$. The division is embedded in $K_{\mathrm{D\_Q}}$. $K_{\mathrm{D\_Q}}$ is the *derivative gain*.

We imposed some conditions on the equations above. 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. However, 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.

Thus, Equation 1 rewrites as:

$$rate_{\mathrm{cgi}}(t+1) = \begin{cases} rate_{\mathrm{cgi}}(t) & \text{if } (\#\text{ hits }) < 0.9 * rate_{\mathrm{cgi}}(t) \wedge \\ & CPU\_util(t) < CPU\_util_{\mathrm{ref}} \\ rate_{\mathrm{cgi}}(t) + K_{\mathrm{P\_CPU}} * e(t) & \text{otherwise} \end{cases}$$

We impose a similar condition on the SYN controller and the bandwidth controller. Both adapt the rates based the queue lengths. The listen queue and the queue to the network interface usually have a length of zero. This means, that the length of the queue 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. Thus, 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.

An important decision is the choice of the sampling rate. For ease of implementation, we started with a sampling rate of one second. To obtain the current CPU utilization, we can use the so-called *jiffies* that the Linux kernel provides. Jiffies measure the time the CPU has spent in user, kernel and idle mode. Since 100 jiffies are equivalent to one second, it is trivial to compute the CPU utilization during the last 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. This is an acceptable solution since the experiment in Section 3.3 shows that the control mechanisms still adapt quickly when we expose the server to sudden high load.

The number of packets queued on an outgoing interface can change quite rapidly. When the queue is full, packets have to be dropped. The TCP connections that experience drops back off and thus less packets are inserted into the queue. We have observed that the queue length has changed from maximum length to zero and back to maximum length within 20 milliseconds. To avoid incorporating such an effect when computing new rates, we sample the queue length to that interface more frequently and compute an average every second. Using this average, the controller updates the rates every second.

3 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., with the default configuration, i.e. a maximum of 150 worker processes. We have extended the length of the web server's listen queue from 128 to 1024. Banga and Druschel [5] argue that a short listen queue can limit the throughput of web servers. A longer queue will not have an effect on the major results.

Parameter Settings

We have used the following values for the control algorithms: The reference values for the queues are set to 100 for the listen queue and 35 for the queue to the network interface which 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. Repeating for example the experiment in Section 3.3 with reference values larger than 20 for the listen queue length leads to the same results. 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. We have obtained this value by experimentation. In our experiments we saw that for gains larger than 1/4, the CGI acceptance rate oscillates between high and low values, while it is stable for smaller values. The proportional gain for the SYN and bandwidth controller is set to 1/16, the derivative gain to 1/4. These values were also obtained by experimentation. When both gains are larger than 1/2, the system is not stable, i.e. the change in both the length of the listen queue and the rates is high when the server experiences high load. When the derivative gain is higher than the proportional gain, the system reacts fast to changes in the queue lengths and the queues rarely grow large when starting to grow. We consider these values to be specific to the server machine we are using[1]. 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.

3.1 Workload

For client load generation we use the sclient traffic generator [5]. 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 [6]:

[1] The values also worked well on another machine we tested, but we do not assume this is the general case.

1. The size of the files stored on the server follows a heavy tailed distribution.
2. The request size distribution is heavy tailed.
3. 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. We separated the 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. Harchol-Balter et al. [23] divide static files into priority classes according to size and assign files larger than 50 KBytes into the group of largest files. We made 20% of the requests for small files dynamic. The dynamic files used in our experiments are minor modifications of standard Webstone [15] CGI files and return a file containing randomly generated characters of the specified size. The fraction of dynamic requests varies from site to site with some sites experiencing more than 25% dynamic requests [22, 21]. 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.

In the next sections we report on the following experiments: In the first experiment we show that the combination of resource-based admission control and TCP SYN policing works and adapts the rates as expected. In this experiment the CPU is the major bottleneck. In the following experiment, we expose the system to a sudden high load and study the behaviour of the adaptation mechanisms under such circumstances. In the experiment in Section 3.4, we make the bandwidth on the interface a bottleneck and show how resource-based admission control can prevent high response times and low throughput. In the last experiment, we show that the adaptation mechanisms can cope with more bursty request arrival distributions.

3.2 CPU Time and Listen Queue Length

In this experiment, we use two controllers: the CPU controller that adapts the acceptance rate of CGI-scripts and the SYN controller. As mentioned earlier, the reference for adapting the rate of CGI-scripts is the CPU utilization and the reference for the TCP SYN policing rate is the listen queue length. About 20% of the requests are for dynamic files (CGI-scripts). In the experiment, we vary the request rate across runs. The goals of the experiment are: (i) show that the control algorithms and in particular resource-based admission control prevent overload and sustain high throughput and low response time even during high load; (ii) show that TCP SYN policing becomes active when resource-based admission control alone cannot prevent server overload; (iii) show that the system achieves high throughput and low response times over a broad range of possible request rates.

For low rates, we expect that no requests should be discarded. When the request rate increases, we expect that the CPU becomes overutilized mostly

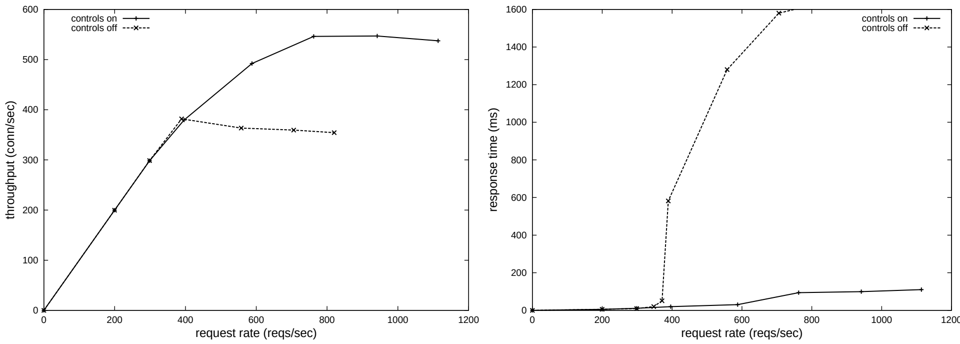

Fig. 4. Comparison standard system (no controls) and system with adaptive overload control

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 thus TCP SYN policing will become active.

Figure 4 compares 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[2]. 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 can be seen 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. Part of this increase is due to the additional waiting time in the listen queue.

We have repeated this experiment with workloads containing 10% dynamic requests and only static requests. If more requests are discarded using HTTP

[2] For higher request rates than those shown in the graph the traffic generator runs out of socket buffers when no controls are applied.

Table 1. Request rate for which SYN policing becomes active for different fractions of dynamic requests in the workload

fraction dynamic reqs. (%)	req rate for onset of SYN policing (reqs/sec)
20	675
10	640
0	610

header-based control, the onset of TCP SYN policing should happen with higher request rates. The results in Table 1 show 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.

In summary, 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 in order to keep the throughput high and response times low.

3.3 Exposure to a Very High Load

In this experiment we 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 Denial-of-Service (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.

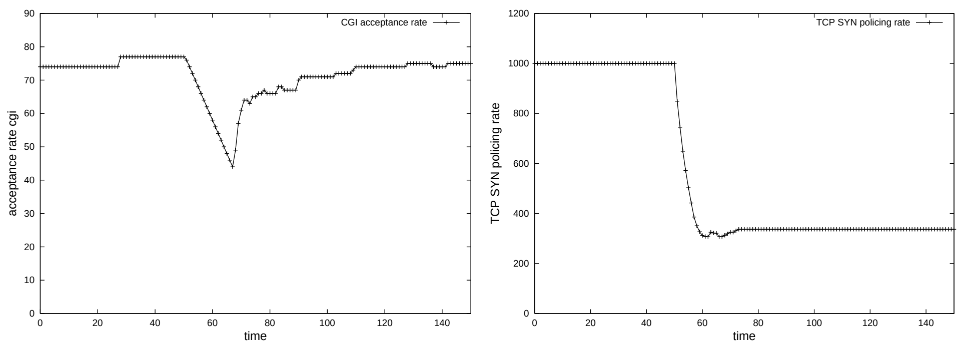

Fig. 5. Adaptation under high load (left CGI acceptance rate, right SYN policing rate)

Figure 5 shows 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. First, since the length of the listen queue increases quickly, the contribution of the derivative part is high. Second, the absolute length of the listen queue is at that time higher than the reference value. Thus, 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. For higher request rates after the period of high load, the SYN policing rate settles at higher rates.

As expected, the CGI acceptance rate does not decrease as fast. With $K_{\mathrm{P\_CPU}}$ being 1/5, the decrease of the rate is at most 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 left-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.

3.4 Outgoing Bandwidth

Despite the fact that the workload used in the previous section contains some very large files, there were very few packet drops on the outgoing 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 in Section 3.2, we can see 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. We therefore 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.

We generate requests to the large file with a rate of 50 and 80 reqs/sec. The results are shown in Table 2. 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 for

Table 2. Outgoing bandwidth

		workload			
		large workload		surge workload	
req rate large workload	metric	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

both workloads is higher using the controls. Note, that when controls are applied the sum of the throughput for both workloads is the same for both request rates (about 331 reqs/sec).

3.5 Burstier Arrival Requests

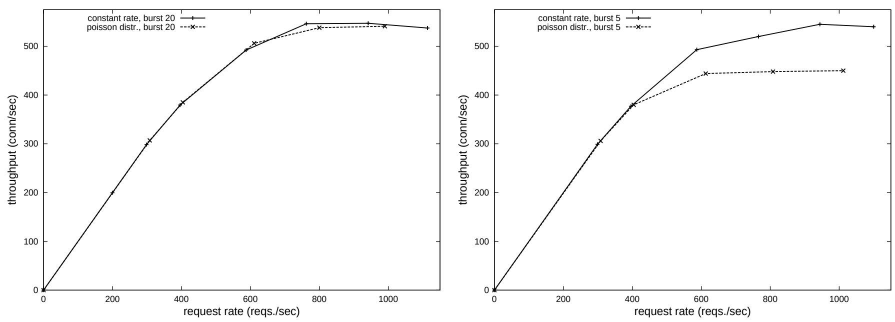

Fig. 6. Comparison between constant request rate and traffic generated according to Poisson distribution for different bucket sizes (left 20, right 5)

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. To verify that our controllers can cope with burstier traffic we have repeated the experiment in Section 3.2 with a Poisson distribution. In Figure 6 we show the throughput at constant rate and at a Poisson distribution. The x-axis denotes the mean of the Poisson distribution or the constant request rate of the standard sclient program while the y-axis denotes the throughput. The difference between the two graphs is that the bucket size of the policers is 20 in the left-hand graph and five in the right-hand graph. Traffic generated at a constant rate should be almost independent of the bucket size since it arrives regularly at the server and a new token should always be available given the token rate is sufficient.

The left-hand graph shows that we achieve about the same throughput independent of the distribution of the requests when the policer's bucket size is 20. If the policer's bucket size is small as in the right-hand graph, more requests than necessary are rejected when the distribution of the requests' arrival times is burstier. This experiment shows that our adaptation mechanisms should be able to cope with bursty traffic provided we make a sensible choice of the bucket size.

4 Architectural Extensions

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 HACC [27] and LARD [26]. In LARD and HACC, the front-end distributes requests based on locality of reference to improve cache hit rates and thus increase performance. Aron et al. [25] increase the scalability of this approach by performing request distribution in the back-ends. In our 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 presented in Section 2.3. 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.

The front-end also computes the rate of the SYN policers for each back-end based on the listen queue lengths reported by the back-ends. Using these values, the front-end itself performs SYN policing, using the sum of the acceptance rates of the back-ends as acceptance rate to the whole cluster. 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.

For a geographically distributed web cluster resource utilization of the servers and the expected resource utilization of the requests can be taken into account when deciding on where to forward requests.

Our architecture is implemented as a kernel module, but could 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. But the same mechanisms could be deployed in user space or in a middleware layer.

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.

It is also straightforward to extend the architecture to handle persistent connections. Persistent connections represent a challenging problem for web server admission control since the HTTP header of the first request does not reveal any information about the resource consumption of the requests that might follow on the same connection. A solution to this problem is proposed by Voigt and Gunningberg [32] where under server overload persistent connections that are not regarded as important are aborted. The importance is determined by the cookies. This solution can easily be adapted to fit our architecture. If a resource is overutilized, we abort non-important persistent connections with a request matching the filter rule associated with that resource.

It would be interesting to perform studies on user perception. Since the TCP connection between server and client is already set up when HTTP header-based control decides on accepting a request, we can inform the client (in this case the user) by sending a “server busy” notification. TCP SYN policing, on the other hand, just drops TCP SYNs which with current browsers does not provide timely feedback to the client. This is another reason for keeping TCP SYN policing inactive as long as resource-based admission control can prevent server overload.

The netfilter framework which is part of the Linux kernel contains functionality similar to TCP SYN policing. We plan to invest if TCP SYN policing can be reimplemented using netfilter functionality. Other operating systems such as FreeBSD contain firewall facilities that could be used to limit the bandwidth to a web server. It is possible to use such facilities instead of SYN policing. However, since there is no one-to-one mapping between bandwidth and requests, it is harder to control the actual amount of requests entering the web server.

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.

5 Related Work

Casalicchio and Colajanni [8] have developed a dispatching algorithm for web clusters that classifies client requests based on their impact on server resources. By dispatching requests appropriately they ensure that the utilization of the individual resources is spread evenly among the server back-ends. Our and their approach have in common that they utilize the expected resource consumption of web requests, however, for different purposes.

Several others have adopted 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 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. [14] use a feedback control approach for guaranteeing relative delays in web servers. Parekh et al. [11] 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. [20] 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. [19] use a mechanism similar to TCP SYN policing to avoid server overload.

Several research efforts have focused on overload control and service differentiation in web servers [3, 7, 13, 9]. *WebQoS* [7] 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. [9] 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 [28] 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.

There are some commercial approaches that deserve mention. Cisco's *LocalDirector* [18] enables load balancing across multiple servers with per-flow rate limits. Inktomi's *Traffic Server C-Class* [17] provides system server overload detection and throttling from traffic spikes and DoS attacks by redistributing requests to caches. Alteon's *Web OS Traffic Control Software* [16] parses HTTP headers to perform URL-based load balancing and redirect requests based on content type to servers.

6 Conclusions

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 under high load.

7 Acknowledgements

This work builds on the architecture that has been developed at IBM TJ Watson together with Renu Tewari, Ashish Mehra and Douglas Freimuth [31]. The authors also want to thank Martin Sanfridson and Jakob Carlström for discussions on the control algorithms and Andy Bavier, Ian Marsh, Arnold Pears and Bengt Ahlgren for valuable comments on earlier drafts of this paper.

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.

References

1. Abdelzaher T., Bhatti N.: Web Server QoS Management by Adaptive Content Delivery. Int. Workshop on Quality of Service (1999)
2. Abdelzaher T.,Lu C.: Modeling and Performance Control of Internet Servers. Invited paper, IEEE Conference on Decision and Control (2000)
3. Almeida J., Dabu M., Manikutty A., Cao P.: Providing Differentiated Levels of Service in Web Content Hosting. Internet Server Performance Workshop (1999)
4. Arlitt M., Williamson C.: Web Server Workload Characterization: The Search for Invariants. Proc. of ACM Sigmetrics (1996)
5. Banga G., Druschel P.: Measuring the Capacity of a Web Server. USENIX Symposium on Internet Technologies and Systems (1997)
6. Barford P., Crovella M.: Generating Representative Web Workloads for Network and Server Performance Evaluation. Proc. of SIGMETRICS (1998)
7. Bhatti N., Friedrich R.: Web Server Support for Tiered Services. IEEE Network (1999) 36–43
8. Casalicchio E., Colajanni M.: A Client-Aware Dispatching Algorithm for Web Clusters Providing Multiple Services. 10th Int'l World Wide Web Conference (2001)
9. Cherkasova L., Phaal P.: Session Based Admission Control: A Mechanism for Improving the Performance of an Overloaded Web Server. Tech Report: HPL-98-119 (1998)
10. Eggert L., Heidemann J.: Application-Level Differentiated Services for Web Servers. World Wide Web Journal (1999) 133–142
11. Parekh S. et al.: Using Control Theory to Achieve Service Level Objectives in Performance Management. Int. Symposium on Integrated Network Management (2001)
12. Glad T., Ljung L.: Reglerteknik: Grundläggande teori (in Swedish). Studentlitteratur (1989)
13. Iyer R., Tewari V., Kant K.: Overload Control Mechansims for Web Servers. Performance and QoS of Next Generation Networks (2000)
14. Lu C. et al.: A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. Real-Time Technology and Application Symposium (2001)
15. Mindcraft: Webstone, http://www.mindcraft.com

16. Alteon: Alteon Web OS Traffic Control Software. http://www.nortelsnetworks.com/products/01/webos
17. Inktomi: Inktomi Traffic Server C-Class. http://www.inktomi.com/products/cns/products/tscclass_works.html
18. Cisco: Cisco LocalDirector. http://www.cisco.com.
19. Jamjoom H., Reumann J.: QGuard: Protecting Internet Servers from Overload. University of Michigan CSE-TR-427-00 (2000)
20. Bhoj P., Ramanathan S., Singhal S.: Web2K: Bringing QoS to Web Servers. Tech Report: HPL-2000-61 (2000)
21. Challenger J., Dantzig P., Iyengar A.: A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. Proc. of ACM/IEEE SC 98 (1998)
22. Manley S., Seltzer M.: Web Facts and Fantasy. USENIX Symposium on Internet Technologies and Systems (1997)
23. Harchol-Balter M., Schroeder B., Agrawal M., Bansal N.: Size-based Scheduling to Improve Web Performance. http://www-2.cs.cmu.edu/ harchol/Papers/papers.html (2002)
24. van de Ven A.: KHTTPd, http://www.fenrus.demon.nl/
25. Aron M., Sanders D., Druschel P., Zwaenepoel W.: Scalable Content-aware Request Distribution in Cluster-based Network Servers. Usenix Annual Technical Conference (2000)
26. Pai V., Aron M., Banga G., Svendsen M., Druschel P., Zwaenepoel W., Nahum E.: Locality-aware Request Distribution in Cluster-based Network Servers. International Conference on Architectural Support for Programming Languages and Operating Systems (1998)
27. Zhang X., Barrientos M., Chen J., Seltzer M.: HACC: An Architecture for Cluster-Based Web Servers. Third Usenix Windows NT Symposium (1999)
28. Aron M., Druschel P., Zwaenepoel W.: Cluster Reserves: a Mechanism for Resource Management in Cluster-based Network Servers. Proc. of ACM SIGMETRICS (2000)
29. Cardellini V., Calajanni M., Yu P.: Dynamic Load Balancing on Web-server Systems. IEEE Internet Computing (1999) 28–39
30. Wang Z.: Cachemesh: A Distributed Cache System for the World Wide Web. 2nd NLANR Web Caching Workshop (1997)
31. Voigt T., Tewari R., Freimuth D., Mehra A.: Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. Usenix Annual Technical Conference (2001)
32. Voigt T., Gunningberg P.: Kernel-based Control of Persistent Web Server Connections, ACM Performance Evaluation Review (2001) 20–25