WebQ: A Virtual Queue For Improving User Experience During Web Server Overload

Bhavin Doshi, Chandan Kumar, Pulkit Piyush, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay
Email: {bhavin,chandan,pulkitpiyush,mythili}@cse.iitb.ac.in

***Abstract*—This paper describes a system for improving user experience when accessing overloaded web servers. While several techniques exist today to build high-capacity web servers, little attention is paid to the fact that servers often crash when faced with transient overload, causing user experience to degrade sharply when incoming load exceeds capacity. Existing overload control mechanisms focus on some form of admission control to protect the web server from overload. However, all such techniques result in user requests failing, either due to timing out or receiving a "service unavailable" message. More importantly, there is no feedback to the frustrated user about when to retry again, leading to adhoc retries. This paper describes WebQ, a system consisting of two web proxies, that together simulate a virtual queue of web requests, and shape incoming load to match server capacity. Users accessing a server protected by WebQ receive a HTTP redirect response specifying a wait time in the virtual queue, and are automatically redirected to the web server upon expiration of the wait time. The wait times are calculated using an estimate of the server's capacity that is computed by WebQ. Users in the virtual queue are provided with a secure cryptographic token, which is checked to guarantee that the user has waited his prescribed time in the queue. We evaluate the ideas of WebQ using a real prototype implementation. Our experiments show that, with WebQ in place, users experience zero server failures and significantly better response times from a web server, even when the peak load is several times the provisioned capacity.**

I. Introduction

The problem of websites "crashing" due to server overload persists to date, despite huge advances in server technologies. Some recent examples include the crash of AT&T's servers due to simultaneous activations from iPhones in 2011 [5], and the overload of the U.S. government healthcare website in 2014 [7]. Server crashes can happen even when the website capacity has been planned well, because websites may sometimes receive an unexpected peak load that significantly exceeds capacity (e.g., when a website is "slashdotted"). Further, even if the peak load can be anticipated, it may be expensive and impractical to provision a website for peak load that occurs only for a short period of time. For example, the online ticketing portal of the Indian Railways is provisioned to serve a few thousand users a minute. However, for a short period everyday when a block of last-minute tickets go up for sale, about a million users visit the website [6]. In such cases, unless an explicit *overload control mechanism* is in place that *protects* the servers, a crash is nearly certain, resulting in website unavailability.

Several solutions have been proposed to address the problem of web server overload (§II). Most overload control solutions involve some form of traffic policing and admission control to protect the web servers, and do not aim to ensure that user requests are eventually served. Requests coming in during the overload period are simply *dropped*, and the user receives some form of a "service unavailable" error message or his connection times out. It is up to the user then to retry such a request, which the user does arbitrarily, further amplifying the load on the server. The end result from the user's perspective is a non-deterministic wait time with no guarantee of eventual service.

This paper presents WebQ, a system to improve user experience when accessing overloaded web servers. The goal of WebQ is to ensure that every request to an overloaded server is *eventually served*, without the user having to resort to adhoc retrying. Our solution (§III) consists of two front-end web proxies, TokenGen and TokenCheck, that together shape incoming load to match server capacity. New requests coming to the website first arrive at TokenGen. This proxy computes a *wait time* for every request, proportional to the amount of overload at the server. TokenGen replies to every request with a HTTP "redirect after timeout" response, that redirects the user to the website after the wait time. TokenCheck is an inline web proxy that intercepts and forwards this request to the web server. The TokenGen proxy also generates a cryptographic token that can be checked by TokenCheck to verify that the client did wait its prescribed duration. Together, the two proxies simulate a "virtual queue" of web requests to an overloaded web server. The proxies do not maintain any per-user state, and rely on aggregate statistics and cryptographic mechanisms to compute and enforce wait times. This stateless queuing of users makes the proxies themselves scalable and robust to overload.

The ability of WebQ to shape incoming traffic significantly depends on having a correct estimate of the server capacity. To this end, TokenGen and TokenCheck monitor the server's response time and goodput to dynamically discover the server's capacity, defined as the load level that maximizes the ratio of its goodput to average response time. Note that WebQ can also work with other capacity estimation algorithms or a manual capacity configuration by the site administrator.

WebQ improves user experience by making response times more predictable, and by eliminating server crashes that occur due to transient overload. When the web server is not overloaded, users are immediately redirected to the server with negligible overhead. During periods of overload, users are informed of their wait time in the queue, are automatically redirected to the web server after the wait time expires, and receive predictable service from the web server once their turn comes up. All this improvement in user experience is

978-1-4673-7113-1/15/$31.00

achieved without modifying the clients or the server. Note that our solution is complementary to numerous techniques that increase the capacity of the web server itself, e.g., load balancing traffic over several replicas. While such techniques improve the capacity of the server itself, our solution improves user experience during those times when the incoming load exceeds server capacity for various reasons.

Experiments with our WebQ prototype (§IV) show that a web server protected by WebQ can easily handle a peak load that is 10× its capacity, with 100% of the arriving requests eventually getting served. Further, after the users wait for a known duration prescribed by WebQ, subsequent server response times are up to 20× lower and have low variability.

II. Related Work

Web server technologies have matured significantly in the past decade. Elson and Howell [10] provide an overview of several techniques that can be used to handle overload at a web server, e.g., Content Distribution Networks (CDNs), or load balancing across replicas. Researchers have also proposed web service architectures that enable effective overload control (e.g., SEDA [19]). Our work is complementary to such techniques. Even the most well designed and provisioned servers can face peak load that exceeds capacity, and WebQ helps web servers deal with this overload gracefully without compromising on user quality of experience.

Various admission control based solutions [18], [11], [17], [19], [9] have been proposed for controlling overload on web servers, where some form of policing of incoming load is used, and excess requests are dropped. While the above systems try to protect the server from overload and guarantee QoS to the admitted requests, the users that are not admitted are not given feedback as to when to retry. On the other hand, WebQ's goal is to improve user experience for those requests that cannot be served immediately, by providing a deterministic wait time and a guarantee of eventual service.

Several researchers have studied the problem of estimating a web server's capacity, in the context of web server provisioning solutions ([20], [16], [13]). While such systems rely on instrumentation at the server, WebQ's simple capacity estimation algorithm treats the server as a blackbox. That said, WebQ can work with any of these methods, and is complementary to this body of work. The idea of automatically discovering server capacity by probing the server's behavior at various load levels has also been used by systems that automate offline server benchmarking [12], [15]. Unlike these systems, WebQ probes for and discovers capacity online.

III. WebQ Design

A. Setup and Assumptions

Use cases. The goal of WebQ is to improve user experience when accessing overloaded web servers. Our solution is particularly useful in the case of multi-tier application servers that serve dynamic HTTP content in response to user requests. In such cases, each user request consumes finite and measurable computational (or other) resources at the web server and at other tiers (e.g., database server). For example, consider the case of a travel portal that lets users check the availability of travel tickets and make reservations. Servers hosting such requests perform significant computation for every user request (e.g., computing the lowest cost schedule across multiple legs of a journey). Therefore, when offered load exceeds capacity of such a server, the response time of the server increases, queued-up requests take longer to complete, causing to server to eventually run out of resources (e.g., socket descriptors) and turn down new user requests with a "service unavailable" message.

Our solution is deployed as a pair of transparent middleboxes between clients and servers, and does not require modifications to either. During periods of overload, WebQ makes clients wait for a predetermined amount of time and shapes incoming traffic to the server, so that clients arrive at the server at a rate that it can handle. WebQ is only useful when server overloads are transient, and average incoming load is below provisioned capacity eventually. WebQ allows web servers to be provisioned for average load instead of peak load, and insulates them from the consequences of bursty traffic patterns. Note that WebQ does not fully solve the overload problem when the incoming load always exceeds server capacity, and can only help delay (but not eliminate) the need for upgrading server capacity. As such, our work is complementary to techniques that scale server capacity, which are more suitable to alleviate persistent overload.

User Acceptance. We assume that users prefer a known wait time in WebQ's virtual queue to non-deterministic wait times, server crashes, and adhoc retrying. Our assumption is grounded in user studies such as [3] that highlight the importance of feedback during long periods of waiting.

Deployment. We envision the WebQ proxies to be deployed as a third party service (in the cloud, perhaps) to which requests are redirected. If desired, WebQ can also be integrated more closely with the server itself. The functionality of TokenGen can be integrated into reverse proxies, load balancers, application layer firewalls, or other middleboxes that vet requests coming to a web server. TokenCheck performs some simple checks before serving every incoming request, and this functionality can be easily integrated with the web server itself. For ease of exposition, we describe both proxies as separate entities.

We assume that HTTP requests are redirected through WebQ by the web site designer using techniques like DNS redirection, much like how some parts of web pages are redirected to and fetched via CDNs. Note that it is not necessary for all web requests to pass through WebQ; the server can choose to redirect only the most resource-intensive ones. For example, a travel portal can host the landing web page that collects information about the planned trip from the user on its regular server farm. Now, after the user fills up his requirements and hits on the "Submit" button, only the subsequent computationally intensive web request can be redirected via WebQ. Note that servers need not commit to using WebQ at all times as well. Servers can choose to redirect requests to our system only during periods of expected overload, e.g., when a travel portal releases a block of deeply discounted tickets or when a university web site releases examination results.

Workload. For ease of exposition, we assume that all requests to the web server are of equal hardness, and consume similar resources at the server. However, our algorithms work even when the workload to the server consists of different types of requests with different relative hardness. In such

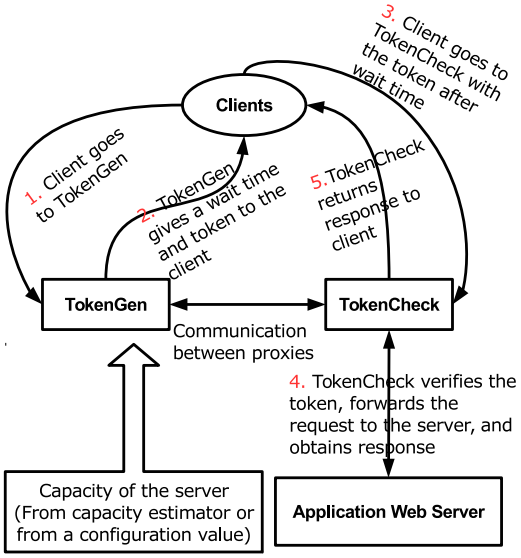

Fig. 1: Architecture of WebQ.

cases, the capacity estimate used by WebQ for traffic shaping will implicitly depend on the relative ratio or *mix* of the different request types. For example, consider a web server that receives two types of requests of different hardness, with their arrival rates in the ratio m:n. Let C be the capacity estimate at WebQ, using which the proxies allow up to C requests/sec to the server. Note that the capacity estimate C implicitly depends on m and n, and would have probably been lower (higher) if the relative proportion of harder requests was higher (lower). Therefore, as long as the incoming traffic to the server adheres to this mix m:n, WebQ's traffic shaping will ensure that the server is not overloaded. When the mix of requests changes, WebQ will perceive this as a change in server capacity, and will rerun its capacity discovery algorithm to discover the capacity corresponding to the new mix. As part of future work, we plan to extend WebQ to better handle the case of varying traffic mix by leveraging several existing techniques to determine server capacity under non-stationary workloads (e.g., [20], [14]).

Overhead. WebQ proxies add a cost overhead to the web server infrastructure. However, if deployed as a third-party service in the cloud with a pay-as-you-go model, website administrators can redirect traffic through WebQ only during periods of expected overload, thereby incurring the cost of the virtual queue infrastructure only when required. Redirection via WebQ will also add an additional network round-trip time to the request completion time. The processing overhead at the proxies itself should be negligible, since the proxies do very little beyond simply redirecting the requests back to the client (when the server is overloaded) or to the server. WebQ shall be deployed when the benefit of improved user experience (during transient overloads) outweighs these costs.

B. Architecture

The WebQ system comprises two entities that work together to simulate a virtual queue: an HTTP proxy server TokenGen that assigns wait times to users, and an inline HTTP proxy TokenCheck that forwards user requests to the web server after the users have waited for the specified time. Figure 1 shows the architecture of our system. User requests that are destined to the web server being protected by WebQ are redirected to TokenGen by the web site designer. TokenGen computes a wait time for requests based on the extent of overload at the server (0 if no overload), and returns a HTTP redirect page to the user that redirects to the web server after the wait time expires. While WebQ uses the HTTP redirect mechanism to make web clients wait, our idea can work with any other mechanism (e.g., a Javascript timer) that can temporarily stall a user from accessing the server.

When the user is eventually redirected to the website, the user's request is intercepted by the TokenCheck inline proxy, and forwarded to the server. In addition to bridging the HTTP connections between the client and the server, TokenCheck also computes statistics about server response time and goodput, and communicates them to TokenGen periodically. TokenGen uses this feedback from TokenCheck to estimate server capacity (§III-E), which in turn feeds into the wait time calculation. Assuming TokenGen calculates server capacity and wait times correctly, the eventual load at the TokenCheck proxy and the web server (after users have waited their prescribed durations) will never exceed the server capacity, even under overload, guaranteeing good quality of experience to the end user.

Note that TokenCheck is protected from overload by TokenGen's traffic shaping, much like how the server is protected. Therefore, it suffices for TokenCheck to handle a load equal to the server capacity and not any more. Therefore any techniques used to scale server capacity can be applied to scale TokenCheck as well. On the other hand, TokenGen may potentially face a much higher incoming load. However, note that TokenGen immediately returns a response to every request, and unlike a traditional inline proxy, does not need to maintain any client sockets open during the duration of the client's interaction with the web server. TokenGen also does not maintain any per client state beyond aggregate traffic statistics. As a result, TokenGen is robust to overload, and can scale to handle a much larger incoming load than the actual web server. For the purpose of this work, we assume that a single TokenGen proxy can handle and redirect all incoming traffic. As part of our ongoing work, we are working on a distributed horizontally-scalable design of TokenGen, where multiple TokenGen replicas perform distributed traffic shaping.

C. Token Generation and Verification

A fundamental question still remains: how do we ensure that the user does not "jump the queue"? For example, the user (or the user's browser) can modify the wait time in the HTTP response from TokenGen, and attempt to access the server sooner than its rightful turn. WebQ uses simple cryptographic mechanisms to discourage such behaviors. TokenGen and TokenCheck share a cryptographic secret key K during setup. When a user arrives at TokenGen, the proxy returns a cryptographic *token* to the user in addition to the wait time. The token is simply the HMAC (hashed message authentication code, computed using the shared secret key) of the user IP address *IP*, the timestamp *TS* when the user checked in at TokenGen, and the wait time w relative to this current timestamp. This token, along with *TS* and w, is embedded in the redirect URL and returned to the client.

When the user arrives at TokenCheck, the proxy extracts the values of *TS*, w, and the token from the redirect URL. The proxy first verifies that the current time matches the sum of the timestamp of the user at TokenGen *TS* and the wait time w prescribed by TokenGen, proving that the user waited exactly for the prescribed time. To verify the authenticity of *TS* and w themselves, the proxy recomputes the HMAC token using the reported values of *IP*, *TS*, and w, and verifies that it matches with the token presented by the user. If the user did tamper with *TS* or w to show up earlier (or later) than his designated time, the HMAC computed by TokenCheck would not match that given to the user by TokenGen. Such non-conforming requests can be dropped by TokenCheck. Note that for the timestamp checks to work as described above, TokenGen and TokenCheck should be time-synchronized. Alternatively, the timestamps can be rounded off to a coarser time granularity to accommodate time drift, without compromising safety.

Note that the timestamp check at TokenCheck also guards against potential replay attacks, where a user reuses old tokens to gain access to the server at a future time. Because the timestamp check verifies that the user has arrived at exactly his designated time, a user that tries to reuse the same token in the future will not be allowed by TokenCheck. It is theoretically possible for a user to reuse his token to gain access to the server multiple times in the short period before the next tick of the timestamp. For example, if timestamps are rounded off to a second, it is possible for the user to reuse the same token multiple times within the one-second interval that was assigned to him for accessing the server. Because TokenGen and TokenCheck do not keep any per-user or per-request state for scalability, such an attack is a possibility. However, because the window of vulnerability is so small (e.g., one second if timestamps are rounded off to a second), we believe allowing a small number of such malicious requests is a reasonable tradeoff for simplicity and scalability of our design.

D. Wait Time Computation in TokenGen

We now describe how TokenGen assigns wait times to requests. TokenGen periodically estimates the capacity of the server (§III-E). The capacity C is a measure of the request-processing capability of the web server, and is measured as the maximum number of requests/sec that the server can successfully handle. The capacity of the server is used as input to compute a suitable wait time for arriving requests if the server is overloaded. The wait time returned to a user indicates the number of seconds the user has to wait before accessing the web server. We only assign wait times in units of seconds (and not milliseconds, for example) for several reasons: (i) the HTTP refresh header supports redirection after an integer number of seconds; (ii) a finer granularity of scheduling is harder to enforce strictly due to network latencies and other delays beyond our control. From now on, we assume that the wait time w returned by TokenGen is an integer and is in units of seconds. However, our design works for any other granularity of wait time that can be reliably enforced.

TokenGen maintains a long circular array of numbers L, where $L[i]$ denotes the number of requests that have been scheduled by WebQ so far to arrive at the web server at a time i seconds into the future. For example, $L[0]$ denotes the number of requests that will be reaching the server in the current second. WebQ can limit the maximum wait time assigned to a client to some large value (say, based on what is considered reasonable from a user's perspective), and the array L can be sized accordingly. Whenever a user request arrives at TokenGen, it finds the earliest timeslot in the future that can accommodate the user, subject to the capacity constraint at the server. That is, it computes the smallest index i such that $L[i] < C$, and assigns the wait time to the user as $w = i$. It also increments the count of requests $L[w]$ by one to account for this user's arrival in the future. Note that if the incoming load is less than server capacity, the wait time will work out to be zero, because $L[0] < C$ always holds. The list L is also updated every second to shift out the previous second's entries.

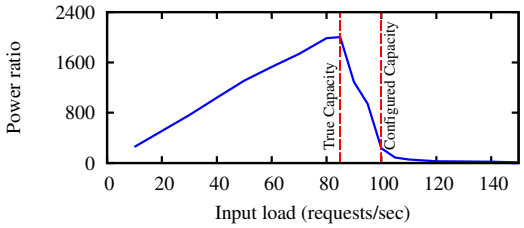

Fig. 2: Power ratio (ratio of goodput to response time) of a web server as a function of offered load.

WebQ also tracks server capacity, and adjusts its capacity estimate from time to time. Changes in server capacity can lead to transient periods where the wait time assignment algorithm deviates from the one described above. Consider the case where TokenGen has scheduled C requests each for the next $T > 0$ seconds into the future, and is currently assigning a wait time of $T + 1$ to new requests. At this time, it discovers that the server capacity has increased, and updates its capacity estimate to $C' > C$. After this update, the wait time assigned to new requests will no longer be $T + 1$, but can be as low as 0, because $L[0] = C < C'$. That is, new requests will be assigned shorter wait times to fill up the newly discovered server capacity in the near future. As a side effect, requests may not always be served on a first-come-first-serve basis during the transient period when capacity is being updated.

Let us now consider the case where the server capacity has reduced and the new capacity estimate $C' < C$. Again, assume that we have already scheduled C requests per second to the server for the next T seconds before we discover the capacity change. Here, we have unwittingly forced the server into an overloaded situation, by scheduling more requests (C) than it can handle (C') for the next T seconds. As a result, the $C * T$ requests scheduled in the future will actually take at least $T' = C * T/C'$ seconds to complete, with $T' > T$. Therefore, new requests that arrive at TokenGen after the capacity reduction are assigned a wait time of $T' + 1$, and no requests are scheduled to the server between T and T'.

E. Capacity estimation

The effectiveness of WebQ crucially depends on assigning appropriate wait times to requests at TokenGen, which in turn depends on knowing the correct capacity C of the web server. We now describe how WebQ estimates this capacity. It is well known that the performance of a web server, as measured by its goodput and response time, degrades significantly when the incoming load is greater than its capacity. For

example, consider a simple web server that is configured to have a capacity of 100 req/s (see Section IV for details of our setup). As the incoming load exceeds capacity, the goodput plateaus off (and eventually drops) and the response time increases sharply. Therefore, the *power ratio*, defined as the ratio of goodput to response time, attains its maximum value around the server capacity, as shown in Figure 2. The peak of the power ratio occurs a little below the configured server capacity because response times of the server start to increase due to queuing even before its configured capacity is reached. We define the *true* capacity (as opposed to the configured capacity) of a server as the offered load (in req/s) that maximizes its power ratio. Our capacity discovery algorithm aims to discover this true capacity, and WebQ shapes incoming traffic to match this capacity in order to keep response times low.

The capacity discovery algorithm collects samples of power ratio at various levels of offered load, builds a function of power ratio vs. offered load in real time, and estimates the value of offered load that maximizes this function. For the purpose of capacity estimation, time is divided into epochs of duration τ (=8 s). Our algorithm *probes* the power ratio at a given value of offered load by scheduling requests using that load level as the capacity estimate in TokenGen for the duration of an epoch. By virtue of being an inline proxy that intercepts all requests to the web server, TokenCheck can monitor the average HTTP response time and goodput of the server during that epoch, and conveys these statistics to TokenGen periodically. Using this feedback, TokenGen computes the power ratio corresponding to the offered load level in that epoch. The capacity estimation algorithm at TokenGen collects several such observations of power ratios by probing different load levels in different epochs, and fits a polynomial curve (degree 3 works well empirically) over the collected samples of power ratio vs. incoming load. The capacity is found as the maximum value of this function. (Note that a certain capacity estimate can be probed only if the incoming load at TokenGen is at least equal to that load level for the duration of the epoch. Our algorithm proceeds with capacity discovery only when incoming load exceeds its current capacity estimate, and pauses discovery during periods of low load.)

Now, how do we pick values of offered load to probe in each epoch? The capacity estimation algorithms starts probing with a low estimate of capacity (15 req/s in our case), and increases this load level by a multiplicative factor in each successive epoch, until a drop in the power ratio is observed. This multiplicative increase helps us to quickly obtain a coarse capacity estimate, irrespective of how low we start relative to the true capacity. Subsequently, the algorithm linearly probes values around this coarse capacity estimate to arrive at an accurate value for the true capacity. Our empirical evaluation shows that the entire algorithm can take a few hundreds of seconds to converge in typical cases. For further details on our algorithm, please see [8].

WebQ also monitors for changes in capacity and adapts to these changes. When the capacity of the server changes, its power ratio curve also changes, and observations of power ratio in subsequent epochs will be far away from the original fitted curve. After capacity discovery completes, WebQ computes the maximum distance between any power ratio observation and the fitted curve, and remembers this maximum error observed during the discovery procedure. If the observed power ratio in any subsequent epoch is at a distance more than twice this maximum error, WebQ empirically concludes that capacity has changed, and reruns its capacity discovery algorithm.

Finally, we note that WebQ can be made to work with any other capacity estimation method also. Alternately, a web service can explicitly specify the rate at which it intends to receive requests (based on its own estimate of its capacity), and WebQ can shape offered load to this specification.

IV. Evaluation

WebQ Prototype. We first describe our prototype implementation. TokenGen is implemented in two parts: the request scheduling logic and capacity estimation. The scheduling logic of TokenGen is implemented as a FastCGI extension [2] to the popular Apache web server. The capacity estimation part of TokenGen is implemented as a separate Java module that keeps listening to the response time and goodput information sent from TokenCheck, and periodically runs the capacity estimation algorithm. Upon change of capacity, it communicates the new capacity estimate to the FastCGI module for scheduling. TokenCheck runs the lighttpd proxy [4]. We modified the proxy code to intercept every request to the web server and perform token verification. The two proxies in our implementation share a 128-bit secret key. The HMAC token is a 128-bit keyed hash (we use MD5, but any other hash function like SHA-2 can also be used). Both proxies use OpenSSL libraries to compute and verify the HMAC token. In our evaluation setup, the TokenGen and TokenCheck prototypes run on separate 4 core Intel i7 desktop machines with 4GB RAM. We find that our unoptimized implementations of TokenGen and TokenCheck are capable of handling over 5000 req/sec each, without any degradation in goodput. The average additional latency due to processing at each proxy was only a few milliseconds.

Experimental setup. Our web server is an Apache installation that runs on a 4 core Intel i7 desktop machine with 4 GB RAM. Client requests to the web server trigger a computationally intensive PHP script that performs integer arithmetic for every request, simulating CPU-bound backend processing in a multi-tier application. The server can be configured to have a certain capacity by suitably adjusting the number of integer operations performed for each request. We simulate client load using Apache JMeter [1] on a 20-core server. The server, clients, and the WebQ proxies are all connected by an uncongested wired network.

Evaluation under transient overload. We now demonstrate the effectiveness of WebQ when servers face transient overload. We configure our web server to have a capacity of 100 req/s, leading to a power ratio peak and true capacity around 80 req/s. We generate an average load of around 600 req/s from the clients for a duration of 4 seconds, and a load of 3 request/sec for the next 32 seconds, such that the average load to the server is below its capacity, but the peak load is much above capacity. The experiment is run for four such cycles, for a total duration of around 150 seconds. Figure 3 shows how WebQ evens out the load to the web server. We can see from the figure that the incoming load at TokenGen is highly bursty. However, due to appropriate scheduling of client arrivals by TokenGen,

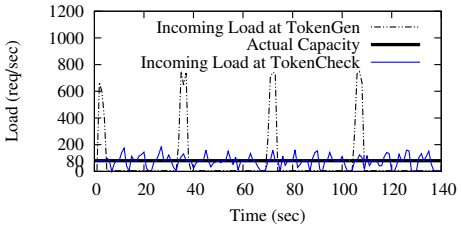

Fig. 3: Traffic shaping of WebQ with bursty incoming load.

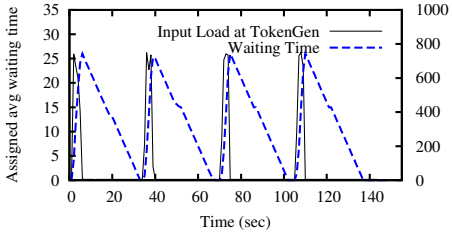

Fig. 4: Wait times assigned by the WebQ TokenGen proxy.

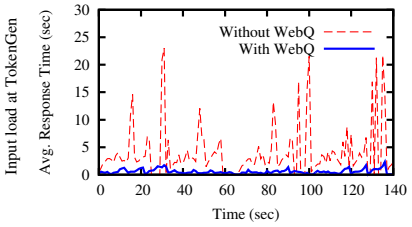

Fig. 5: A comparison of response time of the server with and without WebQ.

the load at TokenCheck (and hence at the web server) is much smoother. Slight fluctuation in the incoming load at TokenCheck is due to the scheduling behavior of various client threads at the client load generator, and is representative of a real deployment where user arrival times may deviate slightly from the assigned wait times due to network delays.

Figure 4 shows the wait times assigned to clients during this experiment. We see that the wait times increase steeply during the burst, forcing clients that arrive during the peak load to wait for longer periods of time. As the incoming traffic burst tapers off, we see that the wait times assigned to clients also become lower. Figure 5 shows the average HTTP response time recorded by clients for their transaction with the web server (as reported by the JMeter tool) with and without WebQ. Note that this response time only counts the time from the moment the redirected request is made to TokenCheck to the time when the server response is returned; it does not include the initial wait time assigned by TokenGen (which was shown in Figure 4). We see that the server response time with WebQ is fairly low (under 1 second) and predictable. That is, once users wait out their time in the virtual queue of WebQ, they can be assured of good service at the server. On the other hand, the response time without WebQ can even go over 20 seconds, and is highly volatile, leading to bad user experience. We omit further details of our evaluation due to lack of space. The interested reader may refer to [8].

V. Conclusion and Future Work

This paper presents WebQ, a system to improve user experience with overloaded web servers. WebQ consists of two proxies, TokenGen and TokenCheck, that together shape incoming load to match server capacity. While most server technologies today focus on improving server capacity, and dropping excess load beyond the capacity, the problem of poor user experience when offered load exceeds this capacity, even for brief periods, hasn't received much attention. Users today face server crashes and connection timeouts when accessing overloaded servers, and resort to adhoc retrying to gain access. In contrast, users of WebQ-protected overloaded servers are presented with a known wait time in a virtual queue of the overloaded server, and are guaranteed service after the wait time expires. With a system like WebQ in place, servers no longer need to be provisioned to handle transient peaks in incoming traffic, eventually leading to cost savings during server provisioning as well.

Our work on WebQ has opened up several exciting avenues for future research. We are developing a capacity estimation algorithm to handle heterogeneous requests with rapidly varying traffic mix. We are also working on a distributed horizontally-scalable design of TokenGen that can scale to handle large loads. Finally, we are exploring the possibility of integrating and testing WebQ with production-quality web servers with capacities of several thousands of req/sec, and under real-life transient overload scenarios.

References

[1] Apache JMeter. jmeter.apache.org.

[2] FastCGI. www.fastcgi.com.

[3] Integrating User-Perceived Quality into Web Server Design. http://logos-software.com/papers/websiteDesign.pdf.

[4] Lighttpd. www.lighttpd.net.

[5] Apple Insider. http://appleinsider.com/articles/11/10/14/rush_of_iphone_4s_activations_forces_some_on_att_into_holding_pattern/, October 2011.

[6] Economic Times. http://articles.economictimes.indiatimes.com/2012-02-23/news/31091228_1_irctc-website-indian-railway-catering-bookings/, February 2012.

[7] USA Today. http://www.usatoday.com/story/news/nation/2013/10/05/health-care-website-repairs/2927597/, October 2013.

[8] Bhavin Doshi, Chandan Kumar, Pulkit Piyush, Mythili Vutukuru. WebQ: A Virtual Queue For Improving User Experience During Web Server Overload. Technical Report TR-CSE-2015-70, Department of Computer Science and Engineering, IIT Bombay, April 2015.

[9] X. Chen, P. Mohapatra, and H. Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proc. of WWW*, 2001.

[10] J. Elson and J. Howell. Handling flash crowds from your garage. In *Proc. of Usenix ATC*, 2008.

[11] H. Jamjoom, J. Reumann, and K. G. Shin. Qguard: Protecting internet servers from overload. Technical Report CSE-TR-427-00, University of Michigan, 2000.

[12] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *Proc. of USENIX ATC*, 2008.

[13] R. Singh, U. Sharma, E. Cecchet, and P. J. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. of ICAC*, 2010.

[14] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Proc. of Eurosys*, 2007.

[15] A. Tchana, B. Dillenseger, N. De Palma, X. Etchevers, J.-M. Vincent, N. Salmi, and A. Harbaoui. Self-scalable benchmarking as a service with automatic saturation detection. In *Proc. of ACM Middleware*, 2013.

[16] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 2008.

[17] R. P. Verlekar and V. Apte. A proxy-based self-tuned overload control for multi-tiered server systems. In *Proc. of HiPC*, 2007.

[18] T. Voigt and P. Gunningberg. Adaptive resource-based web server admission control. In *Proc. of ISCC*, 2002.

[19] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *Proc. of USITS*, 2003.

[20] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni. R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. In *Proc. of Middleware*, 2007.