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 and a simulation model. 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 of the server.**

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.

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 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.

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 and run a simple *capacity estimation algorithm* (§IV) to dynamically discover the server's capacity. Unlike existing server provisioning algorithms [23], [19], WebQ's capacity estimation algorithm treats the server and the service itself as a "black box". It does not rely on any measurements at the server, nor does it require knowledge of customer Service Level Agreements (SLAs) on response times. Instead, WebQ's proxies monitor the average goodput and response time of the server, and schedule requests to the server at a load level that maximizes the *power ratio* (ratio of goodput to response time) of the server as observed at the proxies. Further, the WebQ proxies also detect changes in server capacity and adapt to it quickly; thus our solution works in scenarios where the capacity of the website might be changing with time. Note that while WebQ provides a solution to estimate and track server capacity, it can easily integrate with any other capacity estimation algorithm or with a manual capacity input from the web server administrator.

We have prototyped our solution by modifying existing web proxies (§V). We have also run extensive simulations of our system using existing application-layer simulators. Our experiments and simulations (§VI) 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.

We summarize the contributions of our work as follows:

- A system to improve user experience when accessing overloaded web servers. WebQ consists of a pair of transparent proxies that together implement a virtual queue of web requests and shape incoming traffic to match server capacity. WebQ improves user experience by making response times predictable and by eliminating server crashes.
- A capacity discovery algorithm that identifies the optimal operating point of the server by monitoring its goodput and response times, and adapts automatically in response to changes in server capacity.
- A prototype implementation, a testbed evaluation, and a simulation model to validate our ideas.

II. Related Work

Web server technologies have matured significantly in the past decade. Elson and Howell [12] provide an overview of several techniques that can be used to handle overload at a web server, e.g., Content Distribution Networks (CDNs), load balancing over several replicas, and so on. Our work is complementary to such techniques. Even the most well-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 solutions [21], [14], [13], [20], [22], [11] have been proposed over the last decade for controlling overload on web servers by policing incoming load. Some proposals [21], [13] police incoming TCP SYN packets to limit the number of connections accepted by the server. PACERS [11] uses prioritized scheduling to provide bounded server response times to different classes of requests under overload. A session based overload control scheme was presented in [10], where a dynamic weighted fair sharing scheduler is used to process requests belonging to only those sessions that are likely to be completed. The proxy-based overload control scheme in [20] detects overload when it sees the request arrival rate exceeding the departure rate, and admits new requests only when the server completes earlier requests. WebQ is complementary to all such proposals. 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. Our goal is to improve user experience for those requests that cannot be served immediately due to overload, by providing a deterministic wait time and a guarantee of eventual service.

Researchers have also proposed web service architectures that enable effective overload control. SEDA [22] proposes a staged event-driven architecture for web service design. This architecture allows fine-grained rate limiting at each stage of a multi-tier web service, and enables flexible overload policies. Several researchers have also considered the problem of content adaptation (e.g., [8]), where the quality of the content delivered to the user is reduced to a less resource intensive one when overload is detected. Our work is complementary to such ideas. We focus primarily on shaping user traffic, without assuming any changes to the server architecture or responses delivered to the users.

Several researchers have studied the problem of estimating a web server's capacity. Server capacity estimation is an integral part of recent research on web server provisioning ([23], [19], [16]) that determines the optimal resources to be allocated to a server for a given input load level, such that the end-user SLAs are not violated. However, such systems rely on instrumentation at the server to measure utilization of the bottleneck resource, and assume other knowledge about the server such as the service demands and the response time cutoff to meet SLAs. On the other hand, WebQ's capacity estimation algorithm treats the server as a blackbox and makes no assumptions about server measurement support. That said, WebQ can work with any of these capacity estimation methods, and is therefore 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 server benchmarking [15], [18]. While the capacity discovery algorithm of WebQ bears some similarities to these approaches, WebQ probes for server capacity online unlike server benchmarking systems that operate offline.

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 in the long term. 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. The WebQ proxies can be deployed as a third party service in front of the server being guarded, or can 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, some parts of the paper may 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 cases, the capacity estimated by

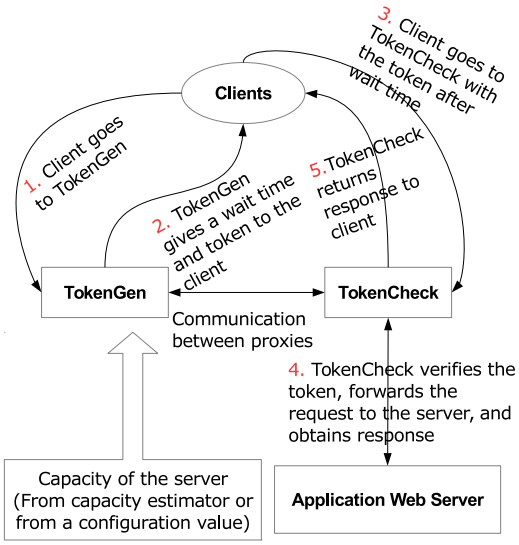

Fig. 1: Architecture of WebQ.

WebQ 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 estimated by 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 with time, WebQ will perceive changes in mix as a change in server capacity, and automatically rediscovers the new capacity corresponding to the new mix. For example, WebQ's capacity estimation algorithm may deduce that the server capacity has reduced if the relative ratio of harder requests increases in the workload. However, because WebQ's capacity estimation algorithm takes a few hundred seconds to converge (§VI-C), we assume that the relative ratios of different request types in the traffic changes infrequently. As part of future work, we plan to extend WebQ to handle the case of rapidly varying traffic mix by leveraging several existing techniques to determine server capacity under non-stationary workloads (e.g., [23], [17]).

Overhead. Redirection via WebQ will add an additional network round-trip time (RTT) to the request completion time. The processing overhead at the proxies itself should be negligible (§VI-A), 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 the overhead of an additional RTT.

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 (§IV), 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 much 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 (§IV). 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

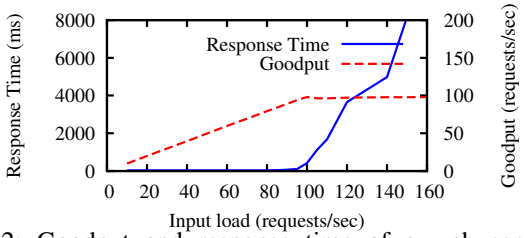

Fig. 2: Goodput and response time of a web server with configured capacity 100 req/s, as a function of offered load.

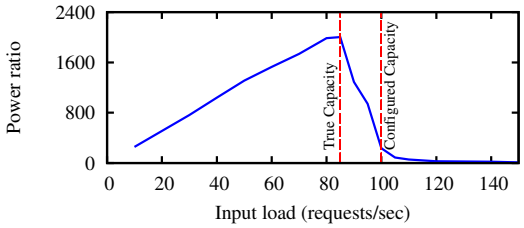

Fig. 3: Power ratio (ratio of goodput to response time) of a web server as a function of offered load.

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.

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 users 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 users will be assigned shorter wait times to fill up the newly discovered server capacity in the near future. As a side effect, users 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 should be assigned a wait time of at least $T' + 1$, and we should not be scheduling any requests to the server for the duration between T and T'.

Note that the above discussion has assumed that all requests to the server are of similar hardness. However, the system will function correctly even if the incoming traffic has different request types, as long as the relative proportion of the different types stays constant or changes slowly with time (§III-A).

IV. 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. WebQ estimates server capacity by monitoring the response times and goodput of the web server, and finds the optimal operating point of the server that is characterized by high goodput and low response times. Our capacity estimate C has an implicit dependence on the mix of the different types of requests in the incoming traffic (§III-A). If the mix of requests changes (resulting in the server becoming capable of processing more or less requests), or if the server capacity changes due to any other reason (e.g., change in configuration), our system will detect a change in capacity, and rediscover capacity. WebQ's capacity discovery algorithm takes a few hundred seconds to converge (§VI-C). Therefore, we assume that server capacity in a WebQ deployment changes infrequently, not more than once every few tens of minutes.

Note that the capacity estimation at WebQ does not assume knowledge of the utilization of the bottleneck resource, service demands, or customer SLAs for response time, that are used by other capacity estimation methods (e.g., [23]). The algorithm only uses as inputs the goodput and average response time metrics that are easily obtained at the WebQ proxies. That said, WebQ can work with any other capacity estimation method as long as the relevant inputs are made available. 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.

A. Key Idea

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 VI for details on the server used in our implementation). Figure 2 shows the average response time and goodput of the server for different values of offered load. The server was run for 60 seconds at each value of offered load to generate this graph. We notice that, as the incoming load exceeds capacity, the goodput plateaus off (and eventually drops) and the response time increases sharply. Given these observations, it is easy to see that the *power ratio*, defined as the ratio of goodput to response time, attains its maximum value around the server capacity. Figure 3 shows the power ratio for the same web server described above, which peaks at a load of around 85 req/s. 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

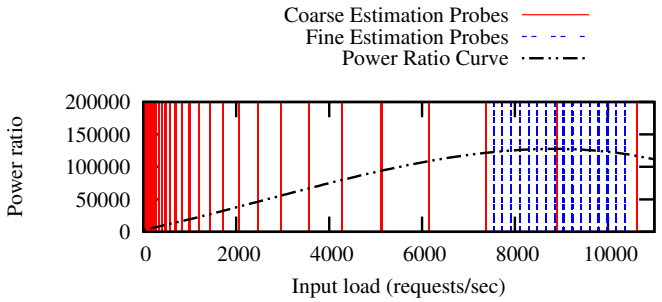

Fig. 4: Capacity probes made to discover the capacity

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. 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. That is, a capacity estimate of c_i at the start of epoch i will cause TokenGen to schedule c_i requests every second for the duration of epoch i. Subsequently, TokenGen computes the power ratio corresponding to the load level c_i using feedback from TokenCheck. 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 all requests. TokenCheck then conveys these statistics to TokenGen periodically. Using this feedback, TokenGen computes average response time r_i and goodput g_i of the web server in epoch i. Let $\gamma_i = g_i/r_i$ denote the power ratio over epoch i. The capacity estimation algorithm collects several such observations of γ_i by probing different values of c_i in different epochs, and fits a polynomial over the curve of power ratio vs. incoming load. The capacity is found as the maximum value of this function. We choose a polynomial of degree 3 to model the power ratio curve, as this function was found to work well empirically.

For ease of exposition, we assume that the incoming load at TokenGen is greater than the capacity of the server, so that TokenGen is actively shaping traffic, and can increase the offered load at the server by increasing its capacity estimate c_i. We will relax this constraint later.

Because capacity estimates are revised once every epoch, the epoch duration should be small to enable convergence to the correct capacity. However, the capacity discovery algorithm relies on the average response time of the server over an epoch duration. Therefore, τ should also be large enough to allow for a reliable estimate of server response times within an epoch. We use $\tau = 8$ seconds in our implementation as it worked well in practice. We leave the auto-tuning of the epoch duration based on the observed variability of response times (much like in [15]) to future work.

B. Phases of Capacity Discovery

Now, how do we pick values of offered load c_i to probe in each epoch? The capacity discovery algorithm is divided into two phases: *coarse estimation* and *fine estimation*. The capacity estimation algorithms starts with a low estimate of capacity c_0 (set to 15 req/s in our case). During the coarse estimation phase, capacity is increased by a multiplicative factor α, i.e., we perform $c_i = \alpha c_{i-1}$ at the start of each epoch. A multiplicative increase in capacity estimates helps us to quickly get close to the true capacity, irrespective of how low c_0 is relative to the true capacity. We use $\alpha = 1.15$.

After every epoch i in the coarse estimation phase, the algorithm checks if capacity has been reached as follows. If the current capacity estimate c_i is less than true capacity, the curve between (c_0, c_i) will be monotonically increasing. On the other hand, once our estimated capacity c_i crosses the true capacity, the power ratio will start to drop, resulting in the peak of the power ratio curve lying in the range (c_0, c_i). The coarse estimation algorithm ends as soon as the peak of the power ratio curve is found in the range (c_0, c_i) for an epoch i, indicating that the current capacity estimate c_i has crossed the true capacity. At this point, the algorithm stops advancing capacity exponentially, and begins a fine-grained search for the true capacity. Let $\bar{C}$ denote the capacity estimate at the end of the coarse estimation phase.

In the subsequent fine estimation phase, the algorithm linearly probes values around the coarse capacity estimate to arrive at an accurate value for the true capacity. At the end of the coarse estimation phase, the offered load to the system is slightly over the true capacity. Therefore, we begin the fine capacity estimation by backtracking our current capacity estimate to a value $c_i = \beta\bar{C}$ with $\beta < 1$. We now start a fine-grained probing for true capacity by increasing our capacity estimate linearly from $\beta\bar{C}$ in small steps of size $c_\Delta = \delta\bar{C}$. We choose $\beta = 0.8$ and $\delta = 0.02$ in our system. The algorithm will continue to search for a global maximum in the current range (c_0, c_i) after every epoch i, much like in coarse estimation.

Once the maximum of the power ratio curve is found in the range (c_0, c_i), the algorithm could terminate and conclude that a capacity estimate has been found. However, to guard the algorithm against an incorrect maximum in the power ratio curve (that may occur before true capacity due to noisy samples in real systems), the algorithm continues to linearly increase capacity for N_f (=7 in our case) extra epochs afterwards. If the maximum of the power ratio curve lies in the range (c_0, c_i) for each of the N_f epochs, then the algorithm concludes that capacity has indeed been found. The capacity estimate C that maximizes the final power ratio polynomial curve fitted over all observations is declared as the true capacity. Figure 4 illustrates how various values of offered load are probed during the coarse and fine estimation phases (from experiments in §VI-D).

C. Refinements

We now describe two modifications to the algorithm above that were added for robustness after initial experiments. First, we note that the server ends up in a transient overload state at the end of the coarse and fine estimation phases, because the algorithm lets the offered load overshoot the true capacity of the server in order to discover a drop in power ratio. The transient overload can last several epochs because TokenGen may have scheduled requests at this higher value of offered load many epochs into the future. Hence, the system needs a brief cool-down period at the end of these two phases to recover from overload and stabilize its

response times. Otherwise, the high response times from the server during this overload period may lead to incorrect power ratio observations at the beginning of the fine estimation phase.

The transient overload of the server at the end of coarse estimation is characterized by the fact that the goodput g_i of the server is higher than the capacity estimate c_i, after the capacity estimate has reduced by a factor β. This happens because TokenGen has scheduled more requests for future slots as per the higher configured capacity before the drop in c_i. Therefore, at the end of the coarse estimation phase, the capacity discovery algorithm waits for the server to *stabilize* by requiring that the goodput g_i be *close* to the capacity estimate c_i for at least N_s (=2 in our case) consecutive epochs. A goodput value g_i is considered to be close enough to c_i if $\frac{|g_i - c_i|}{g_i} < \epsilon$, where ϵ is small (=0.1 in our case). The observed power ratio samples during this transient phase are ignored. A similar cool off period will also exist at the end of fine estimation.

Finally, we note that the above discussion requires that the offered load in the system is greater than the discovered server capacity C for the duration of the capacity discovery. To see why, if the offered load was much below server capacity, TokenGen would not have been able to schedule more than C requests to the server, and would not have observed the reduction in power ratio. Therefore, we modify the algorithm to perform a simple check on offered load during all phases of discovery: the capacity discovery algorithm moves forward (i.e., probes a higher value of offered load) only if the observed goodput g_i is close to the current capacity estimate c_i, where closeness is as defined before. The algorithm with this modification will automatically pause when offered load is not high enough, and resume discovery when offered load goes over the current capacity estimate. Note that no harm is done due to an incorrect capacity estimate when offered load is low because the server is not overloaded (by definition) and WebQ's traffic shaping is not needed.

D. Detecting Capacity Changes

It is not enough if we discover the capacity of a web server once. The web server's capacity can change due to various reasons: change in the traffic mix, provisioning extra capacity, failure of one of the server replicas, and so on. A new capacity must be rediscovered at every such change. One way is to have the administrator or the server management module trigger WebQ's capacity discovery algorithm as needed. In addition, WebQ also has a mechanism to detect capacity 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. If observed power ratios are above the curve, an increase in capacity is inferred, and capacity discovery is triggered with initial capacity set to the current capacity estimate. On the other hand, power ratio points below the fitted curve would indicate a decrease in capacity. In this case, capacity discovery begins at c_0. In both cases, all old power ratio observations are discarded, and capacity discovery proceeds as usual. Note that our algorithm can detect capacity changes only if the change in power ratio is significantly larger than the noise in the samples during capacity discovery. For smaller changes in capacity, it is best if capacity discovery is activated via an external trigger.

V. Implementation

We now 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. We chose the FastCGI option of Apache as it provides ease of implementation without compromising on performance. The TokenGen FastCGI module has three running threads. One thread is used to print debug logs periodically. Another thread communicates with the capacity estimation module for capacity updates. The main thread is responsible for assigning wait times.

Every incoming request at the Apache server running on TokenGen is handed over to the FastCGI module. Apache also passes on additional context about the request via environment variables. The main thread in the FastCGI module then computes the wait time for the request, and returns the appropriate HTTP response to the client. The HTTP response contains the meta HTTP "refresh" header, that automatically redirects the client to the original web server (via TokenCheck) after the prescribed wait time. The redirect URL contains the original URL that the client requested from the application server, along with the following information embedded in the URL string: the current timestamp, the wait time relative to the current timestamp, and a HMAC token that is used to check the authenticity of the reported wait time (§III-C). The entire logic of TokenGen scheduling is under 800 lines of code.

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. The capacity estimation Java module is about 500 lines of code. Note that the decoupled capacity estimation algorithm makes it possible to put a better capacity estimation algorithm or plug a stub that feeds the externally configured values of capacity to the token generation logic in the FastCGI module.

TokenCheck runs the lighttpd proxy [4]. We modified the proxy code to intercept every request to the web server. TokenCheck first strips the timestamp, wait time, and HMAC token information from the requested URL and verifies that the user has arrived at his designated time. If the user's token checks out, TokenCheck makes a request to the web server on behalf of the client, and streams the response back to the client. TokenCheck also has a communication socket open with the TokenGen Java module. TokenCheck sends a notification to TokenGen about arrivals and departures of requests (including the server's response time for completed requests), using which TokenGen can calculate the server's average response time and goodput. We added about 200

lines of code to lighttpd to implement the above changes.

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.

VI. Evaluation

This section presents our evaluation of WebQ. First, we microbenchmark our prototype and show that the proxies add negligible overhead (§VI-A). Next, we show that our WebQ prototype can shape incoming traffic, and significantly improve user experience (§VI-B). With WebQ in place, the server response time was up to 20× lower, and very predictable. Our experiments also show that WebQ can successfully discover server capacity and adapt to changes in capacity within a few hundreds of seconds (§VI-C). Finally, we evaluate WebQ extensively using an application simulator, and show that our results apply to a wide variety of scenarios that were hard to create in the lab (§VI-D).

A. Setup and Microbenchmarks

Our experimental setup consists of several simulated web clients connecting to a web server via the WebQ proxies. We use several high-end desktop machines and a server-blade for our experiments – two desktops run our two proxies, one runs our server, and the server-blade runs our client generator. All components are connected by high speed LAN that was never congested in our experiments.

The TokenGen and TokenCheck prototypes (§V) run on separate 4 core Intel i7 desktop machines with 4GB RAM. We tested each proxy in isolation, without any backend server processing, for varying amounts of incoming client load. 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 TokenGen and TokenCheck is 3 ms and 4 ms per request respectively. We believe that the proxies will be able to handle a much higher load at a lower overhead in an optimized production implementation.

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. Our experiments also require us to simulate variable capacity at the server. To vary server capacity, we use a variable number of server replicas, with the lighttpd proxy in TokenCheck transparently balancing load between them.

We simulate client load using Apache JMeter [1]. JMeter clients send requests to the web server at the specified rate via TokenGen, parse the refresh headers, wait the designated amount of time, and then issue the request to the web server via TokenCheck. We run our client generation script on a 20-core server with 128 GB RAM. While a real user would likely go away and do other useful work when waiting in WebQ's virtual queue, JMeter (and all existing load generators) keeps consuming resources (e.g., threads, memory) for simulated clients even during the intermediate waiting period. As a result, the resource consumption in JMeter was quite high, and it could not sustain a request rate beyond 100 req/s reliably even when running on the most powerful machine in our lab. Therefore, in order to overload the server with this client load generation setup, we configured our server capacity to around 100 req/s (resulting in a power ratio peak around 80 req/s), even though our proxies themselves can handle an order of magnitude higher load. Building a better client load generator that is "WebQ-aware" and does not hold up client request threads in the long waiting phase is part of ongoing work.

B. Overload Control with WebQ

We first test the efficacy of WebQ with respect to smoothing out bursty loads to web servers. 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. After the capacity discovery converges, 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 5 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, 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 and other such issues.

Figure 6 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 7 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. 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.

C. Discovering Capacity

We now evaluate WebQ's server capacity estimation algorithm. For the experiments in this section, the true server capacity that must be discovered is 80 req/s, as indicated by the peak of the power ratio. The offered load from the clients is maintained at around 100 req/s, and the capacity estimates at TokenGen are observed. Figure 8 shows the capacity estimated by WebQ, along with the incoming load

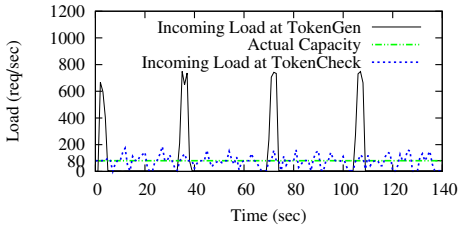

Fig. 5: Traffic shaping of WebQ with bursty incoming load.

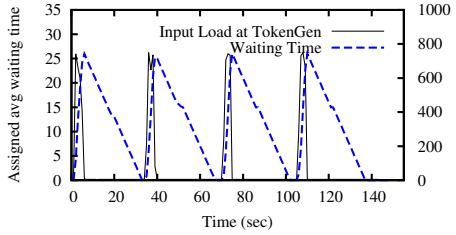

Fig. 6: Wait times assigned by the WebQ TokenGen proxy.

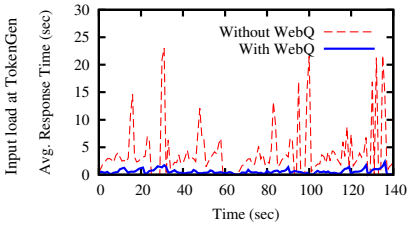

Fig. 7: A comparison of response time of the server with and without WebQ.

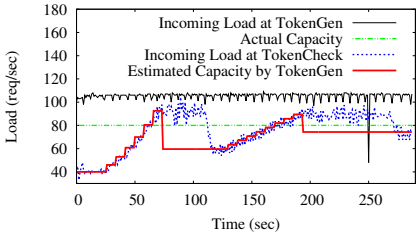

Fig. 8: Capacity discovery in WebQ.

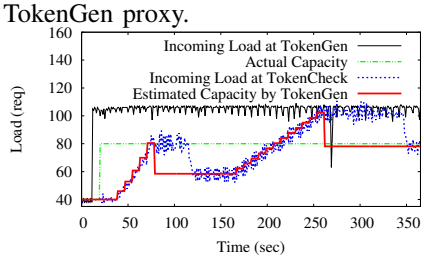

Fig. 9: Handling a capacity increase.

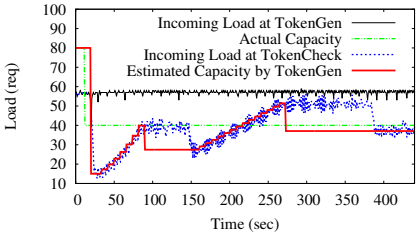

Fig. 10: Handling a capacity decrease.

at TokenGen and TokenCheck. The figure clearly shows the coarse estimation phase with multiplicative increase in capacity, a stabilization phase where capacity discovery halts, followed by the fine estimation phase with linear increase in capacity, and the final true capacity being discovered. The entire capacity discovery process takes about 200 seconds to converge to the correct capacity estimate.

Next, we evaluate whether WebQ can correctly detect changes in server capacity and adapt accordingly. We first increase the capacity of our experimental web server at time $t = 0$ from around 40 req/s to around 80 req/s. Figure 9 shows the capacity estimate of WebQ, along with the actual server capacity, as a function of time. We find that WebQ quickly identifies that server capacity has increased, executes the two phase capacity discovery algorithm, and stabilizes at the higher capacity in under 360 seconds. Note that the time taken by WebQ to adapt its capacity depends on the absolute value of change in capacity that must be explored, starting point of the discovery c_0, the rate of capacity increase α, and epoch size that determines the time interval between capacity increments. As a result, the actual time taken to converge on the new capacity may vary between different deployments.

Finally, we study the performance of WebQ when server capacity reduces. Figure 10 compares the capacity estimate of WebQ with the true capacity of the server. Once again, we find that WebQ correctly latches onto the fact that capacity has reduced, and starts exploring for the server capacity from a pre-configured low enough initial value of $c_0 = 15$. We find that WebQ takes around 420 seconds to converge to the correct capacity in this experiment.

D. WebQ Simulations

Since our experimental setup was limited by the amount of load we can generate in our load generation setup (§VI-A), it was not possible to evaluate WebQ on servers with higher capacities. To overcome this limitation, we modeled and evaluated WebQ's algorithms in an application simulator called PerfCenter [9]. PerfCenter is a discrete-event multi-tier web application simulator, that considers the intricacies of hardware effects and other such deployment information to

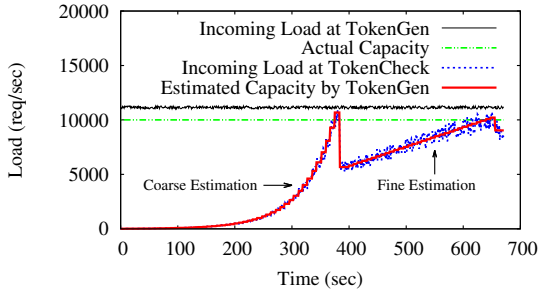

Fig. 11: Capacity discovery of WebQ in simulation.

provide realistic performance estimates. We model a web server protected by the WebQ proxies in this simulator. We ran extensive simulations in this setup to test our algorithms and parameter choices. We present two interesting results from our modeling exercise here.

First, we configured the web server in the simulator to have a normally distributed response time of mean 0.0001 seconds, roughly corresponding to a configured capacity of 10000 req/sec. The server was simulated on a hardware platform with a 40-core CPU. We simulated an offered load slightly over capacity, and observed the performance of WebQ's capacity discovery algorithm. Figure 11 shows the capacity estimates at TokenGen. Note that the stabilization between the two phases is too small to see clearly in this graph. This result shows that WebQ's capacity discovery algorithm works well even at high server capacities.

Next, we test WebQ's traffic shaping mechanism in the case of a rapidly varying traffic mix. Note that WebQ's capacity estimation logic currently does not handle the case where the mix of requests is rapidly changing (§III-A). However, if the server capacity is known by other means, WebQ's shaping logic can schedule the different types of requests suitably to match the capacity. We now demonstrate this scheduling capability of WebQ. For our simulation, we assume two types of requests, Req1 and Req2, with different server response time distributions. TokenGen is configured with the capacity estimate of the server, and the relative

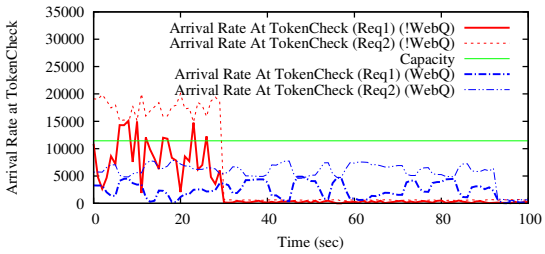

Fig. 12: Traffic shaping with WebQ for bursty incoming load with a variable mix of with heterogeneous requests.

hardness of the request types. As a result, TokenGen can schedule the appropriate number of requests of each type to the server, such that the combined load on the server is below its capacity, even when the mix of the two request types in the traffic changes dynamically. We now load the simulated server with an average load of 30000 req/s for 30 seconds, followed by a quiet period, bringing the average load on the server within its capacity. Figure 12 shows that resulting traffic shaping by WebQ, and the smoothed out arrivals to the server. This result shows that WebQ's traffic shaping mechanism is correctly able to handle a workload of rapidly varying mix of heterogeneous requests.

VII. 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.

While our design is a start in the right direction, our work on WebQ has opened up several exciting avenues for future work, as identified throughout the paper. We are presently working on testing WebQ with real servers with capacities of several thousands of req/sec. We are developing a capacity estimation algorithm to handle heterogeneous requests and rapidly varying traffic mix. We are working on a distributed horizontally-scalable design of TokenGen, where the different replicas exchange information about their scheduled requests in order to perform distributed traffic shaping. We are developing an improved load generator module that addresses the limitations of existing load generators that were observed in our experiments. Finally, we are exploring the possibility of integrating and testing WebQ with a real-life web server.

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] T. F. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks*, 31(11):1563–1577, 1999.

[9] V. Apte and B. Doshi. Powerperfcenter: A power and performance prediction tool for multi-tier applications. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 281–284, New York, NY, USA, 2014. ACM.

[10] H. Chen and P. Mohapatra. Session-based overload control in QoS-aware web servers. In *Proceedings of the IEEE INFOCOM*, volume 2, pages 516–524. IEEE, 2002.

[11] X. Chen, P. Mohapatra, and H. Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proceedings of the 10th International Conference on World Wide Web*, 2001.

[12] J. Elson and J. Howell. Handling flash crowds from your garage. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 171–184, Boston, Massachusetts, 2008.

[13] H. Jamjoom, J. Reumann, and K. G. Shin. Qguard: Protecting internet servers from overload. Technical Report CSE-TR-427-00, Computer Science and Engineering Division, University of Michigan, Ann Arbor, Michigan USA, 2000.

[14] C. Shen, H. Schulzrinne, and E. Nahum. Session initiation protocol (SIP) server overload control: Design and evaluation. In *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, pages 149–173. Springer, 2008.

[15] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *USENIX Annual Technical Conference*, 2008.

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

[17] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, 2007.

[18] 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. 2013.

[19] 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*, 3(1), Mar. 2008.

[20] R. P. Verlekar and V. Apte. A proxy-based self-tuned overload control for multi-tiered server systems. In *High Performance Computing–HiPC*, pages 285–296. Springer, 2007.

[21] T. Voigt and P. Gunningberg. Adaptive resource-based web server admission control. In *Proceedings of Seventh International Symposium onComputers and Communications, ISCC 2002*, pages 219–224, 2002.

[22] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, USITS'03, Seattle, WA, 2003.

[23] 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 *Proceedings of Middleware*, Lecture Notes in Computer Science, 2007.