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

Murali Suresh*, Ravi Shankar Mondal*, Stanly Thomas*, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay
Email: {muralis,ravism,stanly,mythili}@cse.iitb.ac.in

***Abstract*—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 excess user requests failing, and there is no feedback to the frustrated user about when to retry again. 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, without having to maintain per-user state. The design of the WebQ proxies lends itself to a distributed implementation, making the system itself scalable and robust to overload. Our experiments with our WebQ prototype 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 its 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 [4], the overload of the U.S. government healthcare website in 2014 [35], and a recent crash of the Indian e-commerce website Flipkart's servers on a sale day [15]. 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 [10]. 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.

While several solutions have been proposed to address the problem of web server overload, most solutions use some form of traffic policing and admission control to protect the web server itself, 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.

Our previous work proposed WebQ [6], a system to improve user experience when accessing overloaded web servers. While the earlier work only described the basic idea, this paper provides a detailed description of the design, implementation, and evaluation of WebQ. 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 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. Further, both TokenGen and TokenCheck lend themselves to a distributed design, where multiple replicas of the proxy can be used to handle incoming load. This stateless queuing of users, along with a design that lends itself to horizontal scaling, make the proxies themselves scalable and robust to overload.

The ability of WebQ to shape incoming traffic significantly depends on it having a correct estimate of the server capacity. To this end, TokenGen and TokenCheck implement a *capacity estimation algorithm*. The capacity estimation algorithm in WebQ treats the server as a "black box", and does not require any metrics, measurements, or instrumentation at the server to estimate capacity. Instead, the proxies monitor the server's response time and goodput while being on the access path the server, and use these passive measurements to accurately estimate the server's capacity. The server capacity is calculated

as the load level that maximizes the ratio of its observed goodput to the observed average response time (also called the *power ratio*). WebQ's capacity estimation algorithm can accurately estimate capacity across a wide variety of workloads, e.g., when the workload to the server consists of different types of requests with unknown service times in varying relative proportions. Further, the algorithm can detect changes in server capacity from the change in the server's response time and goodput, and use this trigger to periodically re-estimate capacity. Note that WebQ can also work with other capacity estimation algorithms or a manual capacity configuration by the website 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 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.

We have implemented the WebQ proxies by extending existing web servers and proxies. Experiments with our WebQ prototype show that a web server protected by WebQ can easily handle a peak load that is several times 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 significantly lower and have low variability. Our experiments also show that our capacity estimation algorithm estimates server capacity with over 90% accuracy, leading to good traffic shaping by the WebQ proxies.

The contributions of this paper are:

- A novel idea focusing on improving user experience during web server overload. Our idea of shaping user arrivals to match server capacity, by providing a known wait time feedback to users, complements prior work on managing web server overload by increasing server capacity or by performing admission control on the excess traffic.
- A complete design and implementation of WebQ. This paper significantly improves on our prior work [6] that proposed the initial idea, and describes a system that is practical to deploy, robust, and scalable.
- A thorough evaluation demonstrating the efficacy and scalability of our system, under a variety of traffic conditions. Once again, the results in this paper significantly improve upon the basic experiments provided in our previous publication, and show that WebQ works well under a variety of traffic scenarios.

The rest of the paper is organized as follows. Section II describes the high level architecture and design of WebQ. Sections III describes the distributed scalable design and wait time computation in the TokenGen proxy. Section IV describes the capacity estimation algorithm implemented by the TokenGen and TokenCheck proxies. Section V describes our prototype implementation, and Section VI describes our experimental results with this prototype. Section VII discusses related work, and compares our work to prior research. Finally, we conclude the paper in Section VIII.

II. Overview of WebQ

We now describe the architecture of WebQ, and a high-level overview of its various components. We begin with a description of the setup where WebQ is likely to be used, and the assumptions we make about the ecosystem.

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 the 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, the response time of the server increases, queued-up requests take longer to complete, causing the 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 this sense, 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 [16] 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 infrastructure itself. The functionality of the TokenGen module can be integrated

into reverse proxies, load balancers, application layer firewalls, or other middleboxes that vet requests coming to a web server. Similarly, 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 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.

We assume that the WebQ proxies are allowed a small "warm-up" training period before they are expected to be operational. WebQ uses this training period to estimate server capacity and get itself ready to shape traffic. Our capacity estimation algorithm can obtain reasonably accurate capacity estimates within a few minutes (14 minutes as per our evaluation in Section VI) of passively observing traffic to and from the server, without requiring any other information from the server administrator. Web sites using WebQ can plan for this training well before the expected overload. We assume that the training period sees a fairly representative traffic that the web server would see during regular operation.

Workload. We assume that the workload to the web server is heterogeneous, i.e., different types of requests consume different resources at the web server. The mix of the traffic (i.e., the relative proportions of the different request types) is also assumed to be dynamic. We assume that the relative hardness of the requests and the capacity of the server are not known; WebQ estimates these from passive measurements. However, we assume that a request can be classified into one of the few types easily, say, by inspecting its payload or URL. For example, we assume that the WebQ proxies are given enough information to be able to look at a URL and determine if it is a (probably less computationally intensive) request for viewing the availability of tickets on a travel portal, or a (more intensive) request of making the ticket booking, even though the resources consumed by each request are not known. We also assume that the number of different types of requests is finite and not very large (say, of the order of a few tens). During the training period, the capacity estimation algorithm in WebQ tags each request by its type, observes its response time, and uses this information to estimate both the relative hardness of the various requests, and the server capacity.

Note that any change in the resources provisioned at the server, or any change in the server logic to handle a type of request will manifest itself as a change in relative hardness of the requests, and hence a change in server capacity. WebQ's capacity estimation algorithm periodically re-estimates server capacity when it discovers any change. However, because the capacity estimation algorithm takes a few hundreds of seconds to estimate capacity, we assume that the server capacity changes relatively infrequently. Specifically, we assume that the server capacity does not change *during* the training period.

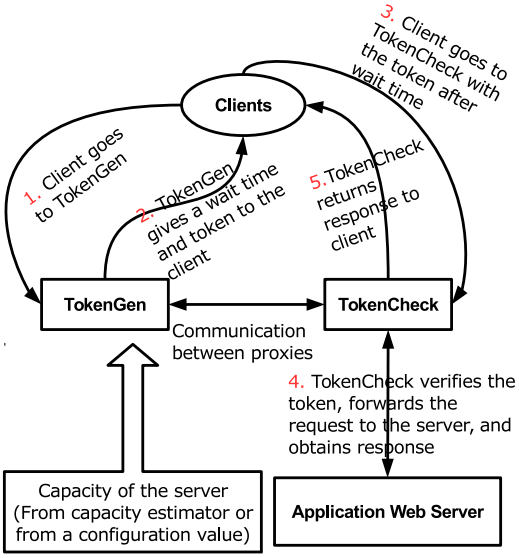

Fig. 1: Architecture of WebQ.

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: a 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. Note that there can exist multiple instances of both TokenGen and TokenCheck, i.e., the designs of the proxies lend themselves to horizontal scaling. However, for ease of exposition, we will first describe the system assuming one instance of each.

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 after the wait time, 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. The capacity estimation module at TokenGen uses this feedback from TokenCheck to estimate server capacity (Section 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 original web server is protected. Therefore, it suffices for TokenCheck to handle a load equal to the server capacity and not any more. As such, any techniques used to scale server capacity can be applied to scale TokenCheck as well. Each TokenCheck instance is stateless, and only relays measurements about server response time to the capacity estimator. Therefore, one can easily have multiple replicas of TokenCheck on the path to the server, balance user traffic across them by directing users to different replicas at TokenGen, and have all replicas communicate with the capacity estimation module independently. Thus the design of TokenCheck is easily scalable and robust to overload.

Achieving a scalable design for TokenGen takes a little more work. The TokenGen proxy consists of two logical components: the forwarding plane that performs traffic shaping and returns a wait time to users, and a capacity estimator that estimates the capacity to shape to. The capacity estimation module only communicates periodically with TokenCheck, and is unlikely to be overloaded. The forwarding plane in TokenGen, however, bears the brunt of the high incoming load to the server, since it is the first point of contact between the heavy user request flow and the web server. Now, 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. Also, TokenGen does not maintain any per client state beyond aggregate traffic statistics. Therefore, TokenGen can scale to handle a much larger incoming load than the actual web server. However, a very high incoming load can still overwhelm a single TokenGen instance. Therefore, the forwarding plane of TokenGen is designed to be horizontally scalable (Section III). Multiple TokenGen replicas divide up the server capacity amongst themselves and shape traffic based on their share of capacity, while ensuring that the wait time that a user sees remains similar no matter which replica the user goes to. This distributed design of TokenGen ensures that WebQ is scalable and robust to overload itself, while ensuring good user experience under overload.

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 must disincentivize such behaviors, while not maintaining significant per-user state (e.g., the wait times allocated to each user) itself. WebQ uses simple cryptographic mechanisms to solve this problem. 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.

III. Wait time computation in TokenGen

We now describe the design of the forwarding plane of the TokenGen proxy of WebQ, which assigns wait times to incoming user requests in order to shape traffic to server capacity. We begin with a simple description assuming a single instance, then generalize our approach to the distributed, scalable design. For ease of exposition, we first assume that all requests to the server are homogeneous, and consume similar resources at the server. We will generalize to the case of

heterogeneous workloads after describing how WebQ handles heterogeneous requests in Section IV.

A. Strawman Approach

Let us begin by assuming that a single instance of TokenGen suffices to handle the entire incoming load to a website. Let C denote the server capacity estimated by the capacity estimation module of TokenGen (described next in Section 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 (assuming a homogeneous workload). The wait time returned to a user by TokenGen indicates the number of seconds the user has to wait before accessing the web server, such that the load to the server never exceeds its capacity. 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 N, where $N[k]$ denotes the number of requests that have been scheduled by WebQ so far to arrive at the web server at a time k seconds into the future. For example, $N[0]$ 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 N 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 k such that $N[k] < C$, and assigns the wait time to the user as $w = k$. It also increments the count of requests $N[w]$ 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 $N[0] < C$ always holds. The list N 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 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 $N[0] = C < C'$. 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'.

B. Distributed Design

A design based on a single TokenGen instance detailed above works fine for moderate incoming load, but can quickly turn infeasible if the server is being flooded with enough traffic to overwhelm TokenGen itself. To prevent TokenGen from crashing under overload, we extend the design of TokenGen to be horizontally scalable. That is, TokenGen will consist of multiple instances, and the user can approach any of them to obtain a suitable wait time when the server is overloaded. The design goal of the distributed TokenGen is that all the replicas together should emulate a single "fat" instance of TokenGen. More specifically, the wait time assigned to each incoming request should be the same, no matter which replica the user request arrives at. Note that we only distribute the forwarding plane of TokenGen that replies to user requests with a wait time; we assume that a single capacity estimator suffices to execute the low-overhead capacity estimation algorithm occasionally.

Each distributed TokenGen replica periodically calculates the *share of capacity* S_i that it is entitled to. A simple approach to calculating a replica's share of capacity would be to divide up the total server capacity equally amongst all replicas. Assuming different replicas can receive different amounts of incoming load, such a simple calculation of S_i would result in overloaded replicas assigning longer wait times than underloaded ones, violating our design goal of being as true as possible to the case of a single TokenGen instance. In order to avoid this problem, the total capacity C is divided amongst the replicas in the ratio of their incoming load. Each replica periodically (every second in our implementation) shares with the other replicas its *current incoming load* L_i (in terms of requests/sec over the past second). Then the share of capacity S_i at each TokenGen is calculated as

$$S_i = \frac{L_i}{\sum_j L_j} * C \quad (1)$$

As an implementation detail, our system reserves a certain minimum capacity at each replica, even if the replica isn't receiving any load currently, in order to let the replica gracefully ramp up when it does start receiving load. Thus the share of capacity S_i is never zero, and never falls below a minimum reserved amount for each replica.

Now, each TokenGen schedules requests at a future time instance using its share of capacity at that point of time. But its share of capacity may come down in the future due to a redistribution of incoming load amongst replicas. Given that a TokenGen cannot "cancel" the requests it has

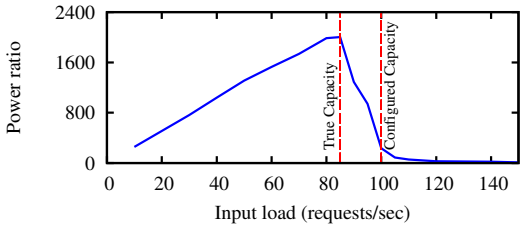

Fig. 2: Power ratio (ratio of goodput to average response time) of a web server, as a function of offered load.

scheduled at some time in the future, all other replicas must honor this excess allocation and allow certain replicas to use more than their share of capacity in a transient period. To enable this, replicas also share with each other the array N_i indicating number of requests each has scheduled to the server in the immediate future. That is, $N_i[k]$ denotes the number of requests the i-th replica has scheduled to the server k seconds into the future. Once all replicas exchange this information with each other, the i-th replica computes the excess allocations for the k-th second into the future $E_i[k]$ as the difference between the actual requests scheduled for the k-th second by all the other replicas and their net share of capacity. Note that the sum of the shares of capacity of all other replicas can be obtained as $(C - S_i)$ at the i-th replica.

$$E_i[k] \quad = \quad \sum_{j \neq i} N_j[k] - (C - S_i) \tag{2}$$

If this excess is positive, the i-th TokenGen replica must accordingly schedule fewer requests than its fair share S_i, to honor the allocations made by the other replicas. Therefore, the i-th replica calculates its *usable capacity* for the k-th second into the future $C_i[k]$ as follows.

$$C_i[k] \quad = \quad \min(S_i, S_i - E_i[k]) \tag{3}$$

Note that, if multiple replicas notice any excess allocation, all of them will reduce their usable capacity by the excess amount, to ensure that the server is not inadvertently overloaded. While this approach is a bit more conservative in its usable capacity estimates than required, it avoids extra communication between the replicas to decide who will accommodate how much of the excess.

Once usable capacity is computed as above, this value (and not the total capacity C) is used to assign wait times to incoming requests as per the algorithm described in Section III-A. That is, when the i-th replica gets a request to access an overloaded server, it computes the smallest index k such that $N_i[k] < C_i[k]$, assigns the wait time to the user as $w = k$, and increments $N_i[k]$ by one to account for this request.

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. We now describe how WebQ estimates this capacity. We begin by describing the key idea intuitively, before proceeding to the formal description of the algorithm.

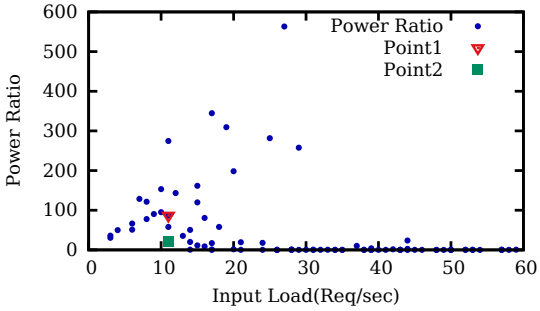

Fig. 3: Average power ratio over 10-second epochs as a function of offered load, with two different requests to the server.

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 V for details of our setup). For now, assume all requests are of similar hardness, i.e., the workload is homogeneous. 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 average 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.

If the workload were homogeneous, the capacity discovery algorithm at WebQ is fairly simple. TokenGen, by virtue of being an inline proxy, observes the requests and their response times at the server. The WebQ proxies could then note the incoming load, goodput, and average response time of the server over some duration of time, plot the power ratio as a function of incoming load, and find the load level that maximizes the power ratio (possibly, by interpolating to unseen load levels by fitting a curve over the power ratio samples).

The above idea of identifying capacity as the incoming load that maximizes power ratio holds only when the workload to the server is homogeneous. What would happen if there were multiple types of requests coming in to the server, each with different service demands at the server? Figure 3 shows a plot of the power ratio in an experiment where a web server was serving two types of requests with different *hardness* values, i.e., the service demands of the two requests were very different. Each point in the plot shows the ratio of the goodput of the server to the average response time (averaged

over an *epoch* of duration 10 seconds) on the y-axis, and the corresponding incoming load (in req/s) on the x-axis. Why does this power ratio plot not have a clean maximum as in the homogeneous case? Consider points 1 and 2 shown in the graph. Both epochs corresponding to these points had the same incoming load (value on x-axis). However, because the *mix* of traffic (i.e., ratio of different types of requests) was different, both epochs saw very different response times from the server, and hence different power ratios. For example, one can infer that the mix in the epoch of point 2 had a greater proportion of "harder" requests than the mix in epoch 1, as seen by the lower power ratio in epoch 2. Because our plot considers each request as one unit, irrespective of its hardness, the goodput and response times of the server will bear no simple relation to the incoming load.

Now, in the figure above, suppose we are told that the second type of request is four times as hard as the first type of request, i.e., if a request of type 1 is equivalent to one "subunit" of work at the server, then a request of type 2 is equivalent to four subunits of work. If this *relative hardness* of the various requests is known, we could plot the power ratio samples a bit differently: we could count the requests not as one unit each, but as a certain number of subunits in proportion to their hardness. If we *scaled* the incoming load, goodput, and response time calculations accordingly, we obtain the plot in Figure 4. In this figure, each point corresponds to the *scaled* power ratio of the server over one epoch. That is, all counts of requests have been scaled by their relative hardness, so that the x-axis indicates the incoming load in subunits, and not just in terms of req/s. Now, from this figure, it is easy to find the number of subunits that maximize the power ratio of the server, and hence its capacity in subunits.

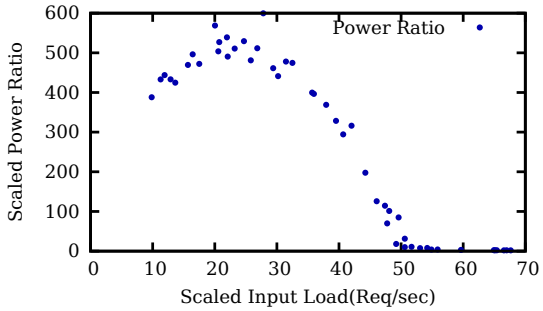

Fig. 4: A scaled plot of the average power ratio of the server, with two different types of requests.

Now, we scaled the plot to make sense by having knowledge of the relative hardness of the various requests to the server. This knowledge is hard to come by in practice. Therefore, WebQ aims to calculate the relative hardness of the various requests itself. How do we calculate the relative hardness? We use the intuition that a correct value of relative hardness will give us a "cleaner" power ratio curve. If relative hardness is correctly estimated, a scaled power ratio plot would have all power ratio samples lying neatly on a polynomial-shaped curve, as in Figure 4. And if the relative hardness were incorrectly estimated, the power ratio curve would be very noisy, like Figure 3. WebQ uses this intuition to search over the space of relative hardness values. Thus, the capacity estimation problem essentially boils down to correctly estimating the relative hardness of various requests to the server, using which the server capacity can be estimated in the subunits that were used to calculate the relative hardness of the requests.

Having described the intuition behind the capacity estimation algorithm, we proceed to describe it informally, followed by a formal description in the next subsection. The capacity estimation algorithm in WebQ runs as part of the TokenGen proxy, at designated *training periods*, either at the start of the deployment, or when changes in capacity have been detected (see Section IV-C). Our evaluation (Section VI) shows that a training period of 14 minutes is enough to obtain reasonably accurate capacity estimates. During the training period, WebQ does not shape traffic, and lets all traffic pass through to TokenCheck with zero wait time. For every request seen at TokenCheck, the proxy relays the request type and its observed response time to the capacity estimation module. (We are assuming that it is possible to easily identify the type of request from its payload, see assumptions in Section II-A). If WebQ uses multiple TokenCheck proxies for scalability, the capacity estimation module will aggregate the feedback from the multiple replicas. Using this training data of request types and response times, the capacity estimation algorithm first tries to estimate the relative hardness of requests. In order to do so, it guesses a value of relative hardness, and scales all observed data by this value of relative hardness. Next, it plots a curve of the scaled power ratio vs. the scaled load using the training data, and fits a polynomial curve over this scaled scatter plot. Using the regression error of this curve fitting as a guide, the capacity estimation algorithm searches over all possible values of relative hardness using gradient descent techniques, to find a value of relative hardness that minimizes the regression error and makes the scaled values fit a curve neatly. Once the relative hardness of all requests has been estimated in some subunits, the capacity of the server (in subunits) is calculated as the value of scaled load that maximizes the scaled power ratio.

Note that the capacity estimation algorithm only estimates capacity and service demands of requests in relative, and not absolute, units. However, this estimation suffices for the purpose of traffic shaping: TokenGen uses this capacity in subunits to shape traffic to the server. Further, every incoming request scheduled by TokenGen would reserve subunits of server capacity proportional to its relative hardness in the future, and this relative hardness of scheduled requests is accounted for by TokenGen in assigning wait times. That is, the variables share of capacity S_i, incoming load L_i, total server capacity C, usable capacity at a replica C_i, and array of requests scheduled N_i would all be accounted for in subunits, and not in req/s as described earlier. With this simple change, the description of the TokenGen wait time calculation holds for the case of heterogeneous workload as well.

Finally, we note that most application web servers in the industry today are multi-tiered, to increase the scalability and reusability of the application architecture. How does WebQ's capacity estimation algorithm apply to a multi-tier server? For a single-tier application, the service time of a request is simply the execution time in a single tier. But in case of multi-tier applications, a request can have different service times in different tiers and it might have conditional jumps that make it traverse different tiers. For example, in a multi-tiered

online retailing application, a browsing request (displaying items in a certain price range) will visit the database tier only once whereas a shopping request (add/remove items from the cart) might have to visit the database tier multiple times with certain visiting probability. However, the basic intuition of our capacity estimation algorithm still holds. In such scenarios, if a request has different service demands by virtue of visiting different tiers probabilistically, the hardness ratios estimated by our algorithm will be the ratios of the *expected* service time of the requests, because we use the average response times in our calculations.

B. Capacity Estimation Algorithm

We now describe our capacity estimation algorithm formally. The input to WebQ's capacity estimation algorithm consists of power ratio samples collected during the training period. We assume that the training period is divided into epochs of duration τ (10 seconds in our implementation). For each epoch, the capacity estimation algorithm accumulates a sample (x, y) on the power ratio plot, where y is the incoming load during the epoch, and y denotes the average power ratio in that epoch.

Let n denote the total number of request types. Let the vector $H = [H_1, H_2, \ldots, H_n]$ denote the vector of relative hardness of the various requests (the exact units of measuring hardness does not matter). That is, let H_i denote the hardness of request of type i (this is assumed to be the constant across all epochs in a given training period). For each epoch, let g_i denote the goodput of the requests of type i, and let m_i denote the relative fraction of requests of type i in the total incoming load of that epoch. Further, let r_i denote the sum of the response times of requests of type i in that epoch. We now calculate the scaled sample (x_s, y_s) for each epoch as follows. The scaled incoming load x_s in the epoch is obtained by summing over all request types, weighing each by its relative hardness and frequency of occurrence in the total traffic:

$$x_s = x * \sum_i (m_i * H_i) \tag{4}$$

Similarly, the scaled goodput for the epoch is obtained as:

$$g_s = \sum_i (g_i * H_i) \tag{5}$$

The scaled average response time is obtained as:

$$r_s = \frac{\sum_i r_i}{\sum_i (m_i * H_i)} \tag{6}$$

The scaled power ratio y_s is thus calculated as:

$$y_s = \frac{g_s}{r_s} \tag{7}$$

Given a training data set, we can scale each point with a guessed value of relative hardness as described above. What values of relative hardness do we use? We guess a value of relative hardness, scale the input data with this relative hardness, fit a degree 3 polynomial curve over the scaled power ratio points, and use the regression error of this curve fitting as a guide to pick the next hardness value. That is, we formulate the relative hardness estimation problem as a constrained optimization problem. We search over the space of all relative hardness values using a gradient descent algorithm. While we use the CMA-ES algorithm [9], any other good gradient descent algorithm should work as well. The gradient descent algorithm tries to pick relative hardness vectors that minimize the regression error of the curve fit. Once the regression error falls below a threshold, we declare that a suitable relative hardness has been found. We then plot the scaled power ratio curve using the final hardness estimate for scaling, and identify the value of scaled input load for which this curve attains a maximum value as the scaled capacity of the server.

See Algorithm 1 for a complete description of our algorithm. We set the initial relative hardness to be 1 for all the requests. We also set the minimum hardness to be 1 and maximum hardness to be 100, in order to constrain the search space and improve running time. Also, we discard scaled power ratio samples that lie below a threshold to obtain a smoother regression curve. To collect input data for the algorithm during the initial training period, we set the server capacity to a very high value, so that the wait time assigned by TokenGen during this period remains zero. During this training period, we assume that web server will observe a representative input load, and WebQ collects several goodput and response time values over several epochs. At the end of the training period, we invoke Algorithm 1 with the collected power ratio samples to estimate relative hardness and capacity (in subunits) of the system. Section VI evaluates the accuracy of this capacity estimation algorithm.

Algorithm 1 Relative hardness and capacity estimation algorithm

1: Input: load vs. average power ratio samples (x, y) for each epoch
2: set minimum value of relative hardness to 1 and max. value to 100. ▷ define the search space
3: set initial hardness ratio to be [1,1,...]. ▷ initialize
4: *regression_error* $\leftarrow \infty$
5: **while** *regression_error* is not minimized **do**
6: Scale each input point (x, y) with current hardness estimate to get scaled (x_s, y_s)
7: Remove the scaled points for which scaled power ratio value is below threshold.
8: Fit a polynomial curve (degree 3) to these points using polynomial regression.
9: Compute *regression_error* as sum of squared difference between the actual values and the predicted output.
10: Make next choice of hardness ratio using CMA-ES.
11: Plot scaled power ratio curve using optimal hardness ratio. Find the value of input load for which it attains the maximum value. This is our estimated capacity (in subunits) of the system.
12: Return hardness ratio and capacity estimate.

C. Handling Capacity Changes

Discovering the capacity only once at the beginning of WebQ deployment may not be enough always. Server capacity

can change for several reasons, including changes in the server application logic or a change in the resource provisioning of the server infrastructure. To detect such capacity changes, WebQ continuously monitors the power ratio samples during the regular run of the system, even after training ends. WebQ then scales these new samples using the previously estimated hardness ratios, and checks if the scaled power ratio values still lie on (or close to) the regression curve obtained during training. WebQ calculates the distance between the new scaled samples and the regression curve, and assumes that the capacity has changed if the distance exceeds a threshold. In our implementation, we set the threshold to be twice the least square regression error of the power ratio curve fitting. Once any capacity change is detected, WebQ starts a training period and reruns its capacity estimation algorithm.

V. Implementation

We begin with a description of our WebQ prototype. We then describe our custom web server load generator we built to evaluate WebQ.

A. TokenGen and TokenCheck

The implementation of TokenGen consists of two parts: the request scheduling logic (or the forwarding plane) and the capacity estimation module. The wait time calculation and request scheduling logic of TokenGen is implemented as a FastCGI extension [12] to the popular Apache web server. We chose the FastCGI option of Apache as it provides ease of implementation without compromising on performance. 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. The entire logic of TokenGen scheduling is under 1200 lines of code.

Each replica of TokenGen is identical, and is configured with the addresses of all the other replicas in the system, in order to communicate and exchange information for distributed traffic shaping. The TokenCheck address to which TokenGen should ask the client to redirect is also configurable. Every TokenGen FastCGI module has $n + 3$ running threads. Where n is the number of peer proxies. One thread is used to print debug logs periodically. Another thread is a listening thread which listens to all the inbound communication towards the TokenGen replica. Inbound communication contains of data from other peer TokenGen replicas and capacity updates from the capacity estimation module. The main thread is responsible for assigning wait times to user requests. The remaining n threads communicate the current wait time data and current incoming load with all other peer TokenGen replicas periodically. Finally, one of the replicas runs the capacity estimation module that provides the total server capacity estimates to all other replicas.

The capacity estimation part of TokenGen is implemented as a separate Java and Matlab module that keeps listening to the response time and goodput information sent from all TokenCheck replicas, and runs the capacity estimation algorithm at the end of the training period. Upon detecting a change of capacity, it runs the capacity estimation again and communicates the new capacity estimate to the FastCGI module for scheduling. The capacity estimation Java module is about 300 lines of code and Matlab module is about 50 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 [22]. 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 capacity estimation module. TokenCheck detects the type of the request from its URL, and sends a notification to TokenGen about arrivals and departures of various type 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.

B. Load Generator Implementation

To test WebQ, we simulate clients sending requests to a server via WebQ, overload the server, and measure the impact of our virtual queue. Since we cannot test with real clients, we must necessarily simulate multiple clients using a web server load generator. When the server is overloaded, the simulated clients must parse the response from TokenGen, extract the wait time and TokenCheck URL from the response, wait for the prescribed wait time, and then issue a request to the web server via TokenCheck. Several existing load generators were found to be inadequate to simulate this test scenario for several reasons, as we describe below, leading us to design and build our own load generator for WebQ.

Multi-threaded load generators like Jmeter [3] spawn a new thread for every client request, and keep the thread alive during the waiting period, unlike a real user who would hopefully do some other useful work during the known wait period. Therefore, simulating an overload situation with non-zero wait times ends up consuming significant resources (CPU and memory) at the load generator, to manage the large number of concurrent live threads. Note that such load generators were primarily built to overload the server with a large number of requests, each of which finished quickly, resulting in a relatively

smaller number of live threads at any point of time. Therefore, several multi-threaded load generators were found unsuitable for our purpose, and could not satisfactorily overload our test web server. Other load generators like Httperf [13] that are single-threaded and use mechanisms like the `select` system call (and not multi-threading) to multiplex various simulated clients were able to sustain enough load. However, such load generators were not rich enough in features (e.g., parsing a HTTP redirect message) for our purpose. Therefore, we have built a custom load generator for WebQ, in order to enable rigorous testing of our prototype.

Our load generator uses light-weight user level threads called fibers from the Quasar core [25] Java library. Our load generator spawns a new user-level thread for each request, which handles the request through its entire lifecycle. However, because multiple user-level threads can be multiplexed on a small number of kernel threads, the actual number of live kernel threads will still be small, because most requests would be in the waiting stage in an overload situation. This design keeps the resource consumption of our load generator much lower than comparable multi-threaded implementations, making it possible to evaluate WebQ under severely overloaded traffic conditions.

Our custom load generator also has several useful features to generate load effectively in our WebQ setup (as well as in other situations where load generators are needed). For example, one feature is a regular expression extractor, which helps in parsing the HTTP response and extracting patterns, which can then be used to create dynamic URLs. Another is a timer which helps in adding delay between requests. In the case of WebQ, the regular expression extractor was useful in extracting out the wait time and TokenCheck URL from the TokenGen response. The constant timer helped in making the simulated user wait before sending the request to TokenCheck. In order to be portable across systems, our load generator is designed and implemented as a web application, using the Spring MVC framework [30]. The user interacts with a front-end web page, which can be used to create and execute complex test plans. A test plan specifies, among other things, the rate at which load to the server must be generated. This request rate in turn controls the rate at which user-level threads are spawned to simulate load in the backend.

VI. Evaluation

We now evaluate WebQ to demonstrate the efficacy of our idea as well as the robustness of our design. We begin with a description of our experimental setup in Section VI-A. Next, we evaluate each component of our system separately. Section VI-B evaluates our distributed traffic shaping mechanism, and shows that a distributed TokenGen system can effectively emulate a single big rate limiter by distributing server capacity in the ratio of the incoming load at each replica. Section VI-C shows that our capacity estimation module can accurately estimate server capacity using only passive measurements, within a few tens of minutes, and with over 90% accuracy. Finally, we put all the components together and evaluate the benefits of the complete system in Section VI-D. We show that a web server protected by WebQ can easily handle incoming load that is several times its provisioned capacity without suffering any crashes. Further, we also show that the response time of the server is significantly lower with WebQ once the user eventually gets served.

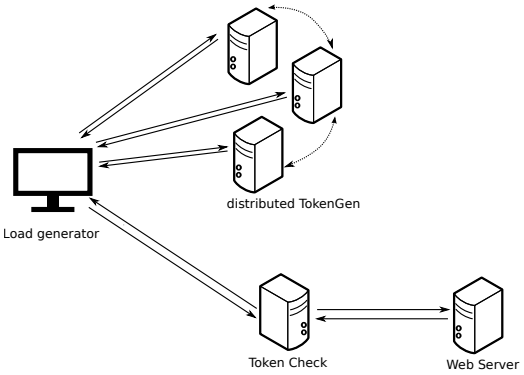

Fig. 5: WebQ evaluation setup.

A. Setup

Our evaluation setup consists of multiple machines running the WebQ proxies, a web server, and a machine running our custom load generator, all connected via a high-speed LAN, as shown in Figure 5. All experiments are run over 3 TokenGen replicas and 1 TokenCheck replica in our distributed WebQ setup, unless otherwise specified. All TokenGen and TokenCheck replicas run on separate 4-core Intel i7 desktop systems with 4GB RAM. Our web server is an Apache installation that runs on a 4-core Intel i7 desktop machine with 4 GB RAM. The Apache installation supports an instantiation of the Moodle [24] course management software, with dummy student and course data. Different client requests to the web server trigger computationally intensive PHP scripts, which query the Moodle database multiple times, simulating CPU-bound backend processing in a multi-tier application. The capacity of the server and the relative hardness of the various requests to the server can be varied by suitably adjusting the number of database operations performed on each request.

We simulate client load using our custom load generator, running on a 40-core Intel server with 128GB RAM. We carefully verified that our client load generation was never the bottleneck in all our experiments. For example, we tested that our load generator can sustain a request rate of 3000 reqs/sec for 60 seconds, even when the the server has a much lower capacity of around 700 reqs/sec. Table I shows the resource consumption by the load generator during this load test. The average CPU and memory utilisations were found to be low, as were the number of live threads.

We also evaluated that the WebQ proxies did not cause a performance bottleneck in any of our experiments. We found that our unoptimized implementations of TokenGen and TokenCheck are capable of handling over 7000 req/sec each, without any degradation in goodput. We believe that an optimized implementation can do much better. Further, we found that each proxy added under 1-2 milliseconds latency to the request processing, an acceptable overhead in our opinion.

	CPU usage(%)	Memory usage(MB)	Number of live threads
Maximum	2.29	1400	86
Minimum	0.0025	105	48
Average	0.15	799	71

TABLE I: Resource Usage by Load Generator.

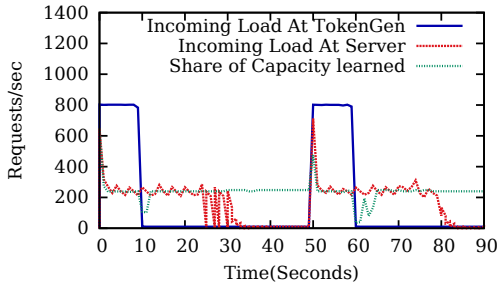

Fig. 6: Load Profile at TokenGen0.

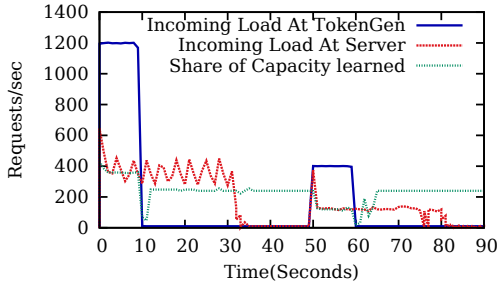

Fig. 7: Load Profile at TokenGen1.

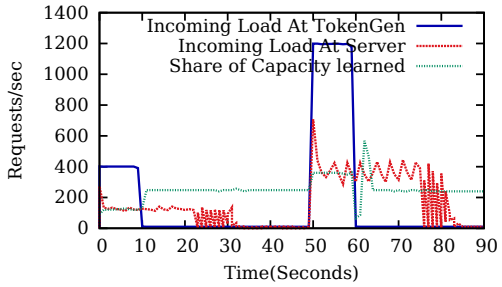

Fig. 8: Load Profile at TokenGen2.

B. Distributed Traffic Shaping

We now evaluate the correctness of our distributed traffic shaping. We configure our web server to have a capacity of 720 req/s. We use a homogeneous workload consisting of requests of a single type in this experiment. We use the load generator to generate an average load of around 2400 req/s for a duration of 10 seconds, followed by a load of 10 request/sec for the next 40 seconds, such that the average load to the server is below its capacity, but the peak load is much above capacity. The load of 2400req/sec is distributed among three TokenGen replicas: TokenGen[0,1,2]. Figures 6, 7, and 8 show the incoming load to each TokenGen, the share of capacity computed by each replica, and the server load contributed by that replica. We see from the figures that the peak incoming load is split as 800 req/s, 1200 req/s, and 400 req/s among

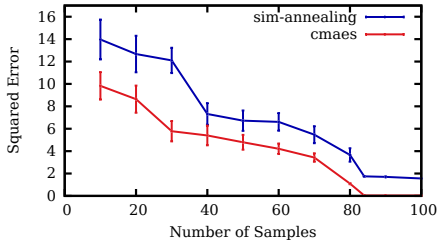

Fig. 9: Squared error in relative hardness vs. number of training samples.

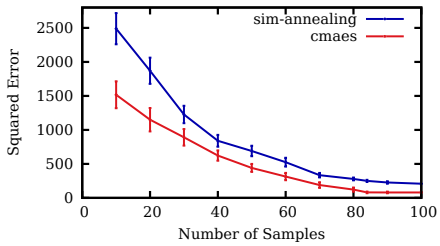

Fig. 10: Squared error in estimated capacity vs. number of training samples.

the three TokenGen replicas in the first 10 seconds of the experiment. From 50sec to 60sec, the load pattern is changed to 800 req/s, 400 req/s, and 1200 req/s respectively. Our results show that the distributed replicas successfully allocate capacity in proportion to the incoming load across the entire experiment. TokenGen[0,1,2] schedule approximately 200 req/s, 400 req/s, and 100 req/s respectively in the first 10seconds, approximately adding up to the configured server capacity of 720 req/s. It can also be noticed from the graphs that the distributed replicas automatically adjust their shares of capacity from 50 sec to 60 sec, in response to the changed input load pattern.

C. Capacity Estimation

We now evaluate the accuracy and effectiveness of WebQ's relative hardness and capacity estimation algorithm. For the experiments in this section, we use Moodle server as the backend. First, we begin with two different types of requests with service times 23ms and 97ms respectively. That is, the "ground truth" relative hardness is 1:4.2 and the capacity of the server is 140 subunits/sec. Now to test the performance of the capacity estimator, we use our load generator to generate a workload trace with random incoming load and changing mix of requests for about 20 minutes. Each value of load lasted for the duration of an epoch (10 seconds), resulting in around 120 power ratio samples over 20 minutes. After collecting the trace, we ran the capacity estimation algorithm multiple times, varying the number of training samples provided as input, to find the optimal number of samples required for an accurate capacity estimate. For each value of the number of samples, we ran our estimation algorithm on 10 different subsets from the collected trace and then computed average squared error and standard deviation of the estimation. We tried two different optimization algorithms—CMA-ES [9] and simulated annealing [19]—in our algorithm, and compared their performance in terms of number of samples required

to reach a certain accuracy. Figure 9 shows the average squared error and standard deviation in the hardness estimate with increasing number of samples for both the optimization methods. Similarly, Figure 10 shows the average squared error and standard deviation in the capacity estimate. From these two figures, we can notice that CMA-ES requires lesser number of samples than simulated annealing for the same accuracy in capacity estimation. With 84 training samples, CMA-ES estimates relative hardness as 1:4.4, and the capacity as 149 subunits/sec. That is, the estimated capacity and relative hardness values have an acceptable error of around 6% and 4.7% respectively. As each sample was collected over an epoch of 10 seconds, this level of accuracy would have required a training period of almost 14 minutes. We believe that training periods of a few tens of minutes would work in most cases.

Next, we evaluated the accuracy of WebQ's estimation algorithm using three different types of requests with service times of 23ms, 50ms, and 97ms respectively. We found that, with 89 training examples, CMA-ES estimates capacity with an error of 7.24%. Results with larger number of request types were similar.

Finally, to test our algorithm in a bigger multi-tier server setup, we enhanced our backend Moodle server to add an extra PHP-based middle tier between the Apache web server tier and the MySQL database tier. We installed the different tiers on different machines. We generated a heterogeneous workload to the server, comprising of two different types of requests that visited the different tiers probabilistically, to emulate a realistic scenario. Finally, we ran the capacity estimation algorithm on the collected power ratio samples. We found that the relative error in hardness estimation was around 4% and the error in the capacity estimate was around 6%, proving that our algorithm can work in a realistic multi-tier setting as well.

D. Impact on User Experience

Having evaluated the individual components of WebQ, we now present results showing the benefits of the complete system. We run an experiment where the web server receives (through a system of three TokenGen proxies) a net load of 2400 req/s for 10 seconds, followed by 10 req/s for 40 seconds and this cycle is repeated. The server has a capacity of 720 req/sec, which is much lower than the peak load. Figure 11 shows how WebQ smooths out incoming load to the 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.

Figure 12 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. It can also be noticed from the graph that all the distributed TokenGen replicas assign similar wait times, realizing our design goal of wait time not depending on which replica a user is redirected to.

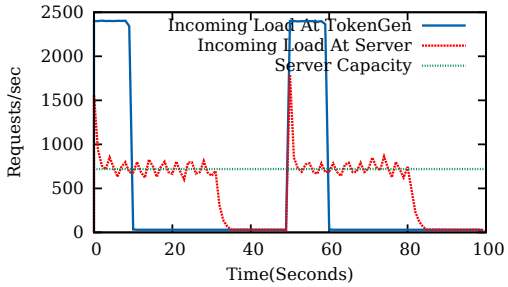

Fig. 11: Traffic shaping of WebQ with bursty incoming load.

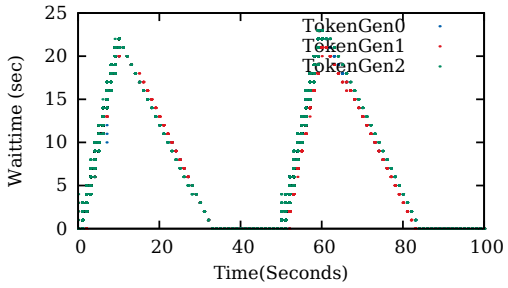

Fig. 12: Wait times assigned by the WebQ TokenGen proxies.

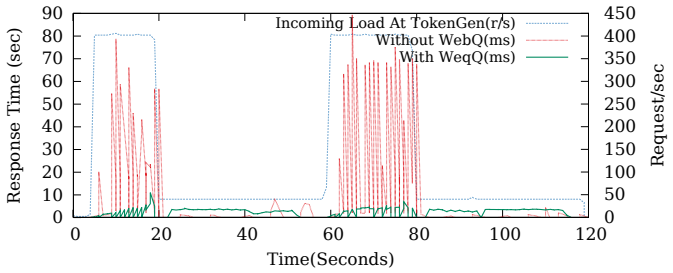

Fig. 13: A comparison of response time of the server with and without WebQ.

Figure 13 shows the average HTTP response time recorded by clients for their transaction with the web server, with and without WebQ. This experiment used a lower peak load (400 req/s) and proportionally lower capacity than what was used in the previous experiments, because the previous value of peak load caused the server to crash in the experiment without WebQ. (The experiment with WebQ ran fine even with the higher load, as expected.) Note that the response time in this figure 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 12). We see that the server response time with WebQ is fairly low (around 5 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 60 seconds, and is highly volatile, leading to bad user experience.

VII. Related Work

Web server technologies have matured significantly in the past decade. Elson and Howell [11] 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. For example, SEDA [39] proposes that web servers should be designed as multiple event-driven stages connected by queues, with rate limiters at each stage to manage overload. 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 admission control based solutions [37], [18], [36], [39], [8] 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. Some capacity planning and scheduling based overload control mechanisms [7] have also been proposed, where a historical trace of requests is used to plan the available capacity. The work in [7] also gives importance to sessions than requests, and is a good design choice as the number of successful sessions completed determines the revenue of a website. Various content adaptation based overload control mechanisms [1], [2] have been proposed, where the quality of the content is degraded to accept more requests under overload. Such content adaptation based methods are less relevant today, with most websites already using optimized content based on load. While all the above solutions are focused on better serving users that are admitted during overload, WebQ's goal is somewhat complementary: to improve user experience for those requests that cannot be served immediately, by providing a deterministic wait time and a guarantee of eventual service.

The distributed TokenGen implementation of WebQ builds upon extensive literature on distributed traffic shaping. The guiding principle for a good distributed traffic shaping mechanism was put forward by Raghavan et al. [26] as " Flows arriving at different limiters should achieve the same rates as they would if they all were traversing a single, shared rate limiter." This principle was closely adhered to in this work. Our work is closest in spirit to the idea of Global Random Drop [26], where the authors use distributed limiters to emulate a central rate limiter. In this work, each limiter estimates the traffic demand locally and later updates a certain number of randomly chosen limiters with their local rates. The number of other limiters to talk to is determined by the branching factor of the gossip protocol. Each limiter keeps track of global demand by aggregating the updates received from the individual limiters. If the estimated global limit exceeds the limit specified, the packets are dropped at a rate proportional to the excess global demand. Several other solutions [31], [27], [5], [23], [21], [17] have also been proposed for the problem of distributed resource limiting. Stanojevic et al. propose C3P [31] where the metric that governs the capacity allocation among the distributed limiters is the loss rate. Some proposals [27], [5] make use of a Token Ring based mechanism to find and assign capacity to each limiter, with the excess requests being dropped or queued with no guarantee on the waiting time. The currency based method presented in [41] deals with distributed rate limiting of static service agreements. WebQ differs from this prior line of work in the sense that it does not deal with rate limiting or dropping of requests; instead it seeks to assign a suitable wait time to requests to shape traffic.

The capacity estimation algorithm of WebQ is also built upon a long line of research on estimating the capacity of web servers and provisioning for it. The method described in [40] gathers CPU utilization at different workload mixes, and then uses least square regression to estimate service demands of different requests. [14] works similarly, but employs Kalman filter instead of regression for estimating the service time of requests. [33] uses low load response time as the service time estimate and then applies Little's law to obtain the capacity of the server. But this model works only for homogeneous workloads. [34] models each tier as G/G/1 queue and the whole system as a closed queuing network, and then estimates the lower bound of capacity for each tier using queuing theory analysis of a G/G/1 queue. [29] uses the same model, but unlike [34], it handles a heterogeneous and changing workload mix. Both [34] and [29] assume that service times of requests are already known to us. WebQ's capacity estimation algorithm differs significantly from all the above, because it is a "black box" method that does not rely on instrumentation at the server to obtain any metrics like CPU utilization or service demands. Instead, server capacity is estimated at the WebQ proxies only based on passive measurements of the server goodput and response time. That said, WebQ can work with any of these methods, and is complementary to this body of work.

While some techniques for blackbox capacity estimation exist, they are not directly applicable in the case of WebQ. The work in [20] uses only server queue length and response time data to estimate capacity. But the underlying assumption in this approach is that service demand distribution is phase-typed and the queuing strategy is FCFS, which is not the case for WebQ. [38] uses Monte Carlo Markov chain method to estimate service demand from queue length samples. But this idea works only for a special type of network called BCMP network. The blackbox server capacity estimation technique in WebQ is different from prior approaches in that it makes no assumptions on the workload or queuing model at the server. 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 [28], [32]. Unlike these systems, WebQ probes for and discovers capacity online.

VIII. Conclusion and Future Work

This paper presented 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 design of WebQ is scalable and robust to overload. Our system can be easily deployed as a third party service, and requires no modifications to clients or servers, beyond a simple redirection of computationally intensive requests via our system under overload by the web server administrator. Experiments with WebQ show that it can correctly shape traffic to match capacity, and significantly improve user experience in the process.

We are in the process of exploring several improvements to WebQ as part of future work. We would like to understand the scalability limits of TokenGen's distributed design, and try to reduce the overhead of communication between the replicas for the purpose of determining their individual shares of capacity. We are also 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] T. F. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks,Elsevier*, 31, 1999.

[2] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *Parallel and Distributed Systems, IEEE Transactions on*, 13, 2002.

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

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

[5] S. Bhatnagar and B. Nath. Distributed admission control to support guaranteed services in core-stateless networks. In *Proc. IEEE INFOCOM*, 2003.

[6] Bhavin Doshi, Chandan Kumar, Pulkit Piyush, Mythili Vutukuru. WebQ: A Virtual Queue For Improving User Experience During Web Server Overload. In *Proc. IWQoS*, June 2015.

[7] H. Chen and P. Mohapatra. Session-based overload control in QoS-aware web servers. In *Proc. IEEE INFOCOM*, 2002.

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

[9] CMAES. https://www.lri.fr/~hansen/cmaesintro.html.

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

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

[12] FastCGI. www.fastcgi.com.

[13] Github - httperf/httperf: The httperf http load generator. https://github.com/httperf/httperf.

[14] X. Huang, W. Wang, W. Zhang, J. Wei, and T. Huang. An automatic performance modeling approach to capacity planning for multi-service web applications. In *Proc. Quality Software (QSIC)*, 2011.

[15] Indian Express. http://indianexpress.com/article/business/business-others/big-billion-day-sale-flipkart-faces-social-media-backlash-over-crashes-misleading-pricing/, October 2014.

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

[17] S. Jamin, P. B. Danzig, S. J. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated service packet networks. *IEEE/ACM Transactions on Networking (TON)*, 5(1), 1997.

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

[19] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6), 1984.

[20] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating service resource consumption from response time measurements. In *Proc. of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, 2009.

[21] A. Kumar, S. Jain, U. Naik, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proc. ACM SIGCOMM*, 2015.

[22] Lighttpd. www.lighttpd.net.

[23] S. Lima, P. Carvalho, and V. Freitas. Handling Concurrent Admission Control in Multiservice IP Networks, 2006.

[24] Moodle. https://en.wikipedia.org/wiki/Moodle, April 2016.

[25] Parallel universe. http://docs.paralleluniverse.co/quasar/.

[26] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *ACM SIGCOMM Computer Communication Review*, 2007.

[27] H. Shao, H. Cheng, and X. Chen. Performing admission control concurrently in core-stateless networks, 2009.

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

[29] 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.

[30] Spring framework - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Spring_Framework.

[31] R. Stanojevic and R. Shorten. Fully decentralized emulation of best-effort and processor sharing queues. In *Proc. ACM SIGMETRICS*, 2008.

[32] 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.

[33] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[34] 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*, March 2008.

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

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

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

[38] W. Wang, G.Casale. Bayesian service demand estimation with Gibbs sampling . In *Proc. MASCOTS*, 2013.

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

[40] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni. A capacity planning framework for multi-tier enterprise services with real workloads. In *Proc. 10th IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[41] T. Zhao and V. Karamcheti. Enforcing resource sharing agreements among distributed server clusters. In *Proc. IPDPS*, 2002.