Proactive Hot Spot Avoidance for Web Server Dependability

Pascal Felber [*]
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

Tim Kaldewey, Stefan Weiss
Institut EURECOM, Sophia Antipolis, France
{*tim.kaldewey, stefan.weiss*}*@eurecom.fr*

Abstract

Flash crowds, which result from the sudden increase in popularity of some online content, are among the most important problems that plague today's Internet. Affected servers are overloaded with requests and quickly become "hot spots." They usually suffer from severe performance failures or stop providing service altogether, as there are scarcely any effective techniques to scalably deliver content under hot spot conditions to all requesting clients. In this paper, we propose and evaluate collaborative techniques to detect and proactively *avoid the occurrence of hot spots. Using our mechanisms, groups of small- to medium-sized Web servers can team up to withstand unexpected surges of requests in a cost-effective manner. Once a Web server detects a sudden increase in request traffic, it replicates on-the-fly the affected content on other Web servers; subsequent requests are transparently redirected to the copies to offload the primary server. Each server acts both as a primary source for its own content, and as a secondary source for other servers' content in the event of a flash-crowd; scalability and dependability are therefore achieved in a peer-to-peer fashion, with each peer contributing to, and benefiting from, the service. Our proactive hot spot avoidance techniques are implemented as a module for the popular Apache Web server. We have conducted a comprehensive experimental evaluation, which demonstrates that our techniques are effective at dealing with flash crowds and scaling to very high request loads.*

1. Introduction

Flash crowds, which result from the sudden increase in popularity of some online content, are among the most important problems that plague today's Internet. This phenomenon, sometimes called the "slashdot effect" [1] after the name of a major technology-oriented Web site, frequently happens when a Web server that usually experiences moderate traffic is linked to by a very popular server. The source server is overwhelmed by simultaneous requests from a large number of clients for which it was not adequately provisioned, and quickly becomes a "hot spot." A server under hot spot condition has its bandwidth and/or processing capacity saturated and typically exhibits faulty behavior, ranging from rather benign performance failures to crashes or even arbitrary behavior. It most frequently stops providing service altogether, as happened to major media companies during the events of 9/11.

While there exist technical solutions to deal with such surges in traffic, they usually involve over-provisioning the service for peak demand and are prohibitively expensive given the infrequent, temporary, and unpredictable nature of flash crowds. Non-commercial sites and small companies cannot justify the cost of such solutions and need cheaper alternatives to survive through transient bursts of massive request traffic.

The goal of this work is to design and evaluate collaborative techniques for quickly offloading the source (or primary) server once the formation of a hot spot is detected. When the traffic reaches a pre-defined threshold for a given Web resource, the primary server replicates the resource on peer servers and subsequently redirects the traffic to the copies using a configurable load-balancing strategy. The servers cooperate in a peer-to-peer manner in that each of them contributes to, and benefits from, the service by sharing the load during periods of high traffic. The proposed techniques are "proactive" in the sense that content replication is performed when detecting a traffic surge and anticipating a flash crowd; they are meant to *avoid* the actual creation of hot spots, rather than heal them. They can be described best as a poor man's approach to increasing the scalability of a Web server by capitalizing the idle bandwidth of other servers.

This paper makes the following contributions. First, we perform an in-depth analysis of the flash crowd phenomenon from the perspective of a Web server. This study shows that flash crowds are very sudden events with an extremely high magnitude, and that only a few resources hosted by the Web server are actually requested by the clients during a flash crowd. Second, we propose practical techniques to anticipate the appearance of hot spots, and to proactively divert the traffic off the primary source. These techniques capitalize the common bandwidth of many Web servers organized in a peer-to-peer

[*] This work was performed while the author was at Institut EURECOM.

manner, in order to sustain a flash crowd while continuing providing service. Third, we present an implementation of our proactive hot spot avoidance mechanisms as a portable module for the popular Apache Web server. This implementation operates transparently to the Web server and can be deployed easily. Finally, we evaluate the effectiveness of our techniques by conducting experiments under high load conditions and under reproduced real-life flash crowd events.

The rest of this paper is organized as follows: We first discuss related work in Section 2. We then study the modelization and detection of flash crowds in Section 3 and introduce our hot spot avoidance mechanisms in Section 4. Section 5 discusses our Web server module implementation, and Section 6 presents results from the experimental evaluation. Section 7 concludes the paper.

2. Related Work

Web server scalability has been an active field of research over the past few years. Some systems have been explicitly designed to support massive demand and degrade gracefully under high load. For instance, the Flash Web server [18] combines an event-driven server for access to cached workloads with multi-threaded servers for disk-bound workloads, to reduce resource requirements while increasing the capacity of the service. SEDA [27] relies on event-driven stages connected by explicit queues that prevent resources to become over-committed when demand exceeds service capacity. The JAWS Web server framework [13] supports multiple concurrency, I/O, or caching strategies to better customize the server to various deployment scenarios. Our approach is different in that we do not try to optimize the Web server *per se*, but rather to improve scalability by capitalizing the common bandwidth and processing resources of several Web servers organized in a peer-to-peer manner (note that the scalability of our techniques will also depend on the performance of the individual Web servers).

Some systems avoid most severe outages by limiting the network traffic upon flash crowds. Several Web servers (e.g., [25]) implement "throttling" mechanisms to limit the number of simultaneous clients, or the bandwidth assigned per client or Web resource; although such mechanisms prevent the server from failing, they provide degraded service to the clients under high load conditions. The NEWS system [7] protects servers from flash crowds by regulating request traffic on the access router, based on observed response performance.

Scalability can also be achieved, albeit at a high cost, through combinations of software and hardware solutions such as traffic load balancers and clusters of servers. For instance, the Google service [12] load-balances queries to one of multiple data centers, each hosting large clusters of machines. Commercial Content Delivery Networks (CDNs) such as Akamai [2] provide scalable Web service by replicating content over a large network of distributed servers, and redirecting clients to local copies using proprietary algorithms added to the domain name system. The most traditional technique to avoid hot spots involves over-provisioning the source servers for peak demand, but given the unpredictable nature of flash-crowds such preventive approaches require a good understanding of traffic patterns. In general, hardware solutions and CDNs have a prohibitive cost and are mostly appropriate when the server is consistently operating under high traffic conditions and hosts critical information.

When dealing with static content, Web caching proxies (e.g., [22]) can be deployed between the servers and the clients (usually at the edge of an ISP) to store content close to the requesting site and make it available to other users [6]. Caching offloads the Web servers, while reducing access time and bandwidth consumption. In [3], the authors evaluate the performance of various multi-level caching techniques to deal with flash crowds. Adaptive caching strategies are discussed in [19]. In practice, however, the cache deployment and configuration policies are locally controlled by individual ISPs or companies and client browsers are often not configured to use Web proxies at all. In addition, for effective caching a proxy must be shared by many clients interested in the same content (the cache hit ratio must be large enough to justify the overhead of the proxy).

Several recent studies have tried to characterize the flash crowd phenomenon [15, 28] and a few proposals have been put forth to leverage peer-to-peer architectures for tolerating such events as well as improving the scalability of Web content distribution. SQUIRREL [14] is a client-side peer-to-peer Web cache that stores recently accessed documents on the client peers, and looks for a reasonably fresh copy in the local caches before fetching a Web resource from the origin server. It does not operate transparently as it requires specific software to be installed on the client computers. PROOFS [24] is a peer-to-peer network that uses randomized overlay construction and scoped searches to efficiently locate and deliver content under heavy demand. When some content cannot be obtained from a server, peer clients try to obtain it from other peers instead. This approach does not heal the hot spots *per se*, rather their symptoms, and it requires specific software to be deployed on the clients. In [23], the authors propose a server-side peer-to-peer Web caching scheme for load balancing client requests under hot spot conditions. The approach is essentially reactive and uses an underlying DHT for storing the content at risk. Our system is most similar to Globule [20] in its implementation. Globule is a Web server module that replicates documents according to observed access patterns, in order to spontaneously create cache copies on peer servers close to the clients. It mostly focuses on improving perceived access latency and overall throughput and operates as a cost-effective CDN; it has not been explicitly designed to cope with flash crowds. In contrast, we perform an in-depth study of real-world flash crowd events, and we specifically analyze and evaluate the applicability of dynamic replication of Web resources to anticipate and gracefully handle such events.

3. Flash Crowds and Performance Failures

To better understand the flash crowd phenomenon, we have studied the log of a server that has been hit by a sudden surge of traffic immediately after being linked on the highly popular *slashdot* Web site (`http://slashdot.org`). The site affected by the flash crowd was hosted on a commercial server, provisioned to cope with reasonably high load; it has not failed, but has quickly become saturated by the data traffic, which typically results in clients experiencing degraded service (e.g., many stalled connections, only a fraction of the requests served). Such a performance failure may not only affect the site hit by the flash crowd and its clients, but may also have an adverse impact on other sites hosted by the same server infrastructure and sharing the same network connections.

The server log covers a period of 28 days in November 2003. It lists $3,936,422$ requests representing a total 51 GB of data sent by the server. The flash crowd occurred on November 18 and lasted about two days. The most severe spike occurred within minutes after the link posting and lasted less than one hour, which tends to indicates that many users were reading *slashdot* at that very time. The traffic remains higher than usual for a few days, probably caused by the users who consult *slashdot* occasionally.

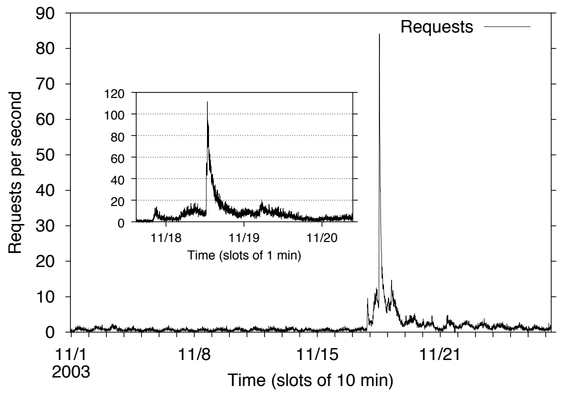

Figure 1. Number of requests per second received by the Web server during flash crowd.

Figure 1 shows the number of requests per second observed in the log during the whole period, averaged over time slots of 10 minutes. We clearly see the impressive spike corresponding to the arrival of the flash crowd, where the traffic increases by two orders of magnitude to reach more that 100 requests per second. This can be observed more accurately in the inner graph, which shows the number of requests averaged over time slots of 1 minute during the three-days period of the flash crowd. The raw log even lists a maximum of 215 requests during the busiest second. Note that the amplitude of a flash crowd is generally independent of the popularity of a Web site: a server that receives only a few requests per days can be hit by a flash crowd of the same magnitude as a popular server, but the latter is usually better provisioned to sustain high traffic.

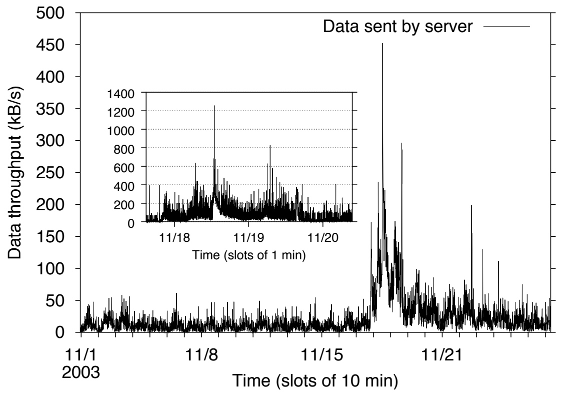

Figure 2. Throughput of data sent by the Web server during flash crowd.

Figure 2 shows the throughput of the data sent by the server averaged over time slots of 10 minutes. We can observe the daytime cycle with less traffic transferred during the night (the site was hosted in the USA, where most of the regular visitors also came from). The flash crowd is clearly visible, with a surge of traffic one order of magnitude higher than usual traffic spikes. When zooming in the flash crowd with time slots of 1 minute, we observe that the data throughput reaches 10 Mb/s, which corresponds to the server bandwidth capacity and effectively saturates its network access links. We have been informed by the site's administrator that he quickly configured the server to stop sending large content to offload the network connection and sustain higher request traffic for small resources. Indeed, the server was also hosting some large software files, seldom requested but representing a non-negligible portion of the instantaneous traffic at times. Such requests for large content are difficult to account for, because the logs do not store information about the download durations and completions necessary for an accurate study of the throughput. We have thus ignored for the rest of the analysis the 0.28% of the requests bigger than 500 kB, as well as the 0.35% of the requests that were downloaded in multiple parts (some clients were downloading large files by requesting different segments in parallel).

We have designed our proactive hot spot avoidance techniques on the premise that only a small number of Web resources (e.g., pages, images) are targeted by a flash crowd. These few resources can then easily be mirrored on the fly to peer servers and traffic redirected to the copies. To validate our hypothesis, we have analyzed the popularity of the $6,705$ distinct resources requested by the clients that appear in the server log (to distinguish between different versions of the same resource, i.e., having same URI, we have considered the size of the server reply). Figure 3 shows the cumulative number of requests for each of the resources sorted by order of popularity. We observe that the request frequency for a resource is inversely proportional to its popularity, with most of the requests targeting only a few pages. A closer look at the most popular requests using logarithmic scales (inner graph) show that the most popular resource (the page linked to by *slashdot*) is requested approximately $85,000$ times during that day, more than twice more frequently than the second one. The resources with popularity ranks between 2 and 23 are also requested very often; they probably correspond to the resources (images) embedded in the linked page, which are typically requested automatically by client browsers. Thereafter, resource popularity follows a classical Zipf distribution.

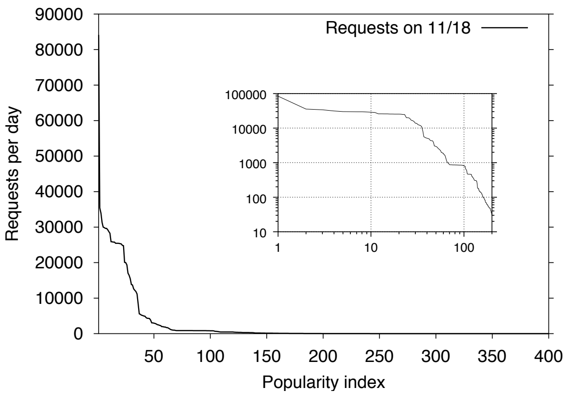

Figure 3. Number of requests for a file upon flash crowd, as a function of its popularity.

Given that the popularity of the resources is highly biased, we have analyzed the request traffic and data volume that the most popular resources were accounting for during the flash crowd. Figure 4 shows that the 100 most popular resources account for 99% of the requests during the highest spike, the top 50 resources for 94%, and the top 30 resources for 84%. Therefore, mirroring only a fraction of the resources can dramatically reduce the number of requests processed by the server.

In terms of data throughput, the most popular resources represent a smaller portion of the total data volume, accounting for 95%, 80%, and 63% respectively (inner graph of Figure 4). This can be explained by the fact that some of the less-requested resources were significantly bigger than the most popular resources. In addition, the Web server logs the entire size of each requested resource even if the transfer does not complete, e.g., when the user interrupts interrupt a lengthy download. In situations where the server is bandwidth-limited and resources have widely different sizes, it might be desirable to focus on the resources that consume most of the network capacity instead of those that are most requested. Thus, there is a trade-off between request processing and bandwidth optimization, and one might use distinct strategies for offloading the server depending on whether the network or the CPU is first saturated.

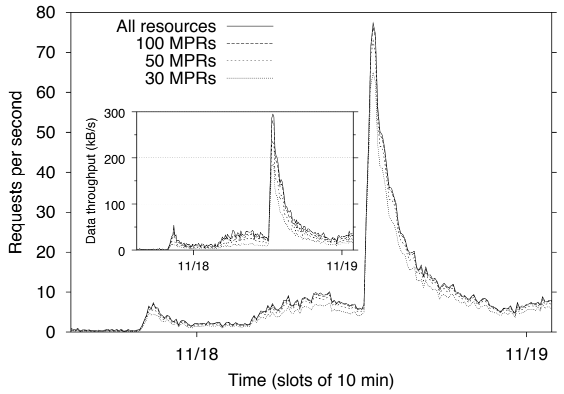

Figure 4. Number of requests per second (outer graph) and data throughput (inner graph) for the most popular resources (MPR).

Interestingly, we observe two small spikes in request traffic and data volume before the actual flash crowd, both when considering all resources and only the most popular ones. This limited activity probably corresponds to the linking of the resources on moderately popular sites. The actual flash crowd that results from the *slashdot* posting occurs 16 hours after the first spike, and more than 4 hours after the second one. This substantiates our intuition that we can often predict a flash crowd in advance by observing traffic patterns and take proactive measure to best sustain it. Considering again Figure 4, it appears that by mirroring the most popular resources and redirecting the clients to the copies, the bandwidth utilization of the primary server would be significantly reduced (given that a redirection message has an empty payload). If HTML pages were "rewritten" while mirroring, i.e., the links to the local resources embedded in the page were modified to refer to mirror copies, the number of requests sent to the primary server would also shrink dramatically (by more than one order of magnitude in the analyzed flash crowd, as the page linked to by *slashdot* contained more than 20 embedded images).

4. Hot Spot Avoidance

The general principle of hot spot avoidance, depicted in Figure 5, essentially consists of four mechanisms. First, the detection of flash crowds achieved by maintaining per-resource hit statistics. Second, the replication of affected content to mirror servers. Third, the redirection of client requests to the replicas. Finally, the return to normal operation once the flash crowd has disappeared. These mechanisms are described in the rest of the section.

4.1. Flash Crowd Detection

As observed in Section 3, a flash crowd is often preceded by smaller spikes of traffic. In order to anticipate the arrival of a flash crowd, we need to keep track of the frequency of requests to the resources hosted by a server, so as to detect which resources are becoming hot spots (e.g., because they are linked to by a very popular site). In general, proactive actions must be taken for those resources only.

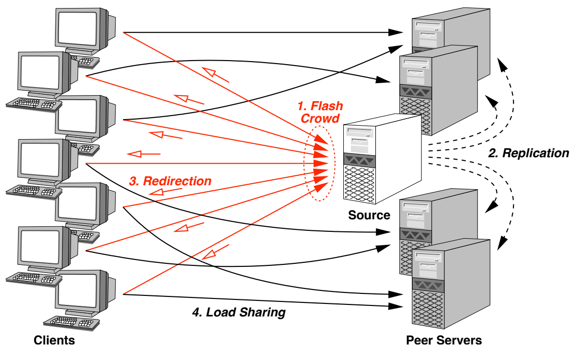

Figure 5. Principle of hot spot avoidance.

To maintain hit statistics for frequently-accessed resources in a space-efficient manner (i.e., without having to keep track of all past requests), we dynamically compute an exponential weighted moving average of the request inter-arrival times, along the same lines as TCP computes its estimated round-trip time. Specifically, we compute an average of the inter-arrival times using the following formula:

$$IAT(x) = (1 - \alpha) * IAT(x-1) + \alpha * (at(x) - at(x-1))$$

We record the arrival time of every hit ($at(x)$) and compute the difference with the previous one. We then combine this value with the previous average inter-arrival time (IAT) to obtain the new average. The constant α is a smoothing factor that puts more weight on recent samples than on old samples and smooths out important variations. Low (high) values of α will increase the stability (volatility) of the moving average. We have used a value of $\alpha = 0.125$, which is also the value recommended for TCP in RFC 2988. Taking into account the uneven popularity of the resources (see Section 3), we maintain individual hit statistics for each resource.

4.2. Resource Replication

Once the arrival of a flash crowd has been detected, the server needs to take proactive measures to avoid becoming a hot spot. It does so by replicating—or mirroring—the most requested content to a set of servers chosen from a pool of cooperating sites. The number of mirror servers necessary to withstand a flash crowd obviously depends on its intensity, as a set of n servers will typically each receive $\frac{1}{n}$ of the requests of the source server. The way servers discover and monitor each other is orthogonal to our problem; it can be achieved by various means, from static configuration files to sophisticated peer-to-peer substrates.

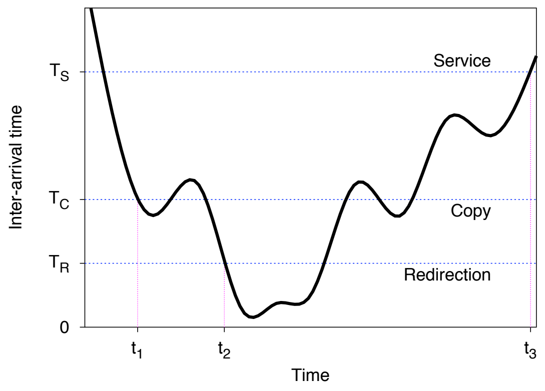

Figure 6. Configurable thresholds.

Resource replication takes place once the average inter-arrival time reaches a pre-defined "copy threshold" T_C (see Figure 6). The copying process may involve rewriting some of the hyperlinks of HTML documents. Replication can be achieved by either pushing content to the mirror server (e.g., using an HTTP PUT or POST request), or by asking the mirrors to pull the content from the source (e.g., using an HTTP GET request). We generally favor the push model, as it gives more control on the transfer process to the primary source. Further, when copying an HTML file, we will preferably also copy all the resources embedded in the file (e.g., images) so that they are requested from the mirror; this will significantly decrease the request rate experienced by the primary source.

4.3. Request Redirection

After a resource has been replicated on mirror servers, the source server can start redirecting clients toward the copies. This is achieved by sending an HTTP response with code 302 indicating to the client that the resource has temporarily moved to another address. When receiving such a reply, client browsers automatically fetch the resource from the address specified in the HTTP message header. The primary server can use simple load sharing policies, such as choosing a mirror at random or following a round-robin strategy, or even more sophisticated techniques, such as estimating the load of the mirrors based on their capacity and the traffic previously redirected. We have found the round-robin strategy to be effective in practice, while imposing very low processing overhead on the primary server.

As we dimension the copy threshold to proactively replicate resources before being network- or CPU-saturated, it is usually not necessary to immediately redirect traffic to the mirrors after the replication process has completed. Therefore, we introduce a second "redirection threshold" T_R, typically twice smaller than T_C, to trigger the actual redirection of requests (see Figure 6). This combination of threshold values enables us to fine tune the proactive (replication) and reactive (redirection) operating modes of the server. Of course, optimal performance can only be achieved if these thresholds are adequately provisioned.

Note that, in case a mirror becomes overloaded with too many redirected requests, it can in turn replicate some mirrored content further to another set of servers. Thus, the whole process can be applied recursively, at the price of additional redirections.

4.4. Return to Normal Operation

After a flash crowd has passed and the hit frequency decreases past a certain threshold, the primary server should return to normal operation, i.e., serve the files directly to the clients. In order to avoid that small oscillations of the average inter-arrival time around the redirection threshold repeatedly activate and deactivate request redirection, we introduce a distinct "service threshold" T_S, typically several times bigger than T_R, to delay the return to normal service (see Figure 6).

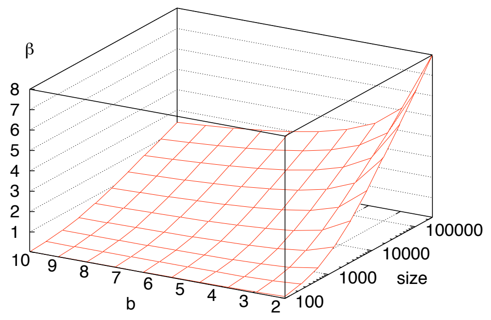

Figure 7. Evolution of the weighting factor β with $k = 1,000$.

Once request redirection has been deactivated for a given resource, the mirror servers can reclaim the local storage associated with the resource's replicas. To that end, we propose a simple distributed garbage collection scheme that requires each server to allocate a limited amount of cache storage for mirrored resources. When the cache is full, the servers remove the least recently accessed resources (note that we could also use cooperative caching strategies, such as *N-Chance Forwarding* [9], to improve the overall cache efficiency). If a mirror receives a redirected request for some content that it no longer holds, it first checks the HTTP request header, which carries the address of the primary server in the "referrer" field. It then replies to the client by redirecting it back to the primary server. When the source receives the request (now redirected twice), it learns that the mirror server specified as referrer does not hold the requested content anymore and must be removed from the list of active mirrors for that resource. The same mechanisms can be used to indicate to the source that a server is not able, or willing, to serve some mirrored content anymore (e.g., because it is becoming overloaded).

4.5. Size-Dependant Triggering

The size of the requested resources plays an important role in the bandwidth usage of the source server. Large resources can consume a significant portion of the network capacity even when requested at a reasonably low frequency. Conversely, very small resources may not even need to be replicated on mirror servers, as they are scarcely larger than a redirection message. It is therefore important to take the size of the resources into account when triggering resource replication and redirection.

Therefore, we introduce a size-dependent weighting factor β: resource replication, redirection, and normal service are triggered when the average inter-arrival time *multiplied by* β reaches the threshold values T_C, T_R, and T_S, respectively. We define β as:

$$\beta = log_b \left(1 + \frac{size}{k}\right)$$

where $size$ is the size of the requested resource, k specifies a constant size reference, and base b controls how much β is affected by size variations. For instance, given $b = 2$ and $k = 10$ kB, a resource of 10 kB will yield a weighting factor $\beta = 1$. Smaller documents will quickly bring its value down close to 0, thus delaying or preventing request redirection altogether. Larger documents will increase the value of β and trigger redirection at lower hit frequencies. Figure 7 illustrates the evolution of the weighting factor β as a function of the base b and the size of the requested resource, with the constant $k = 1,000$ bytes. We can observe that β is more dependent on the size for smaller values of b.

5. Web Server Module

We have developed a prototype hot spot avoidance module in Perl using the mod_perl extension of the Apache Web server. The module maintains hit statistics for each of the recently accessed resources. These statistics include the time of the last access, the estimated average inter-arrival time of requests, and the list of all servers that have a copy of that resource (or an empty list if the resource is not mirrored). Since Apache uses multiple independent child processes to handle client requests, the module maintains a consistent copy of the statistics in a hash table stored in shared memory. Our first version of the prototype keeps both the hit statistics and the mirroring information in shared memory (Figure 8 left), which guarantees that all processes see the same consistent inter-arrival times and reach the different thresholds simultaneously. We observed that the runtime cost of shared memory accesses and mutual exclusion (incurred for every request) led to a noticeable degradation of the performance. Therefore, we also implemented an alternative strategy where the hit statistics are kept in the local memory of the processes and only the mirroring information is stored in shared memory (Figure 8 right). This strategy still guarantees that a resource is replicated only once and by a single process, but it has much lower runtime overhead because the processes need to access the shared memory only when the thresholds are reached. The drawbacks are that processes may start redirecting requests at different times if the load is not equally distributed and the thresholds have to be dimensioned so as to take into account that each process will only see a fraction of the requests.

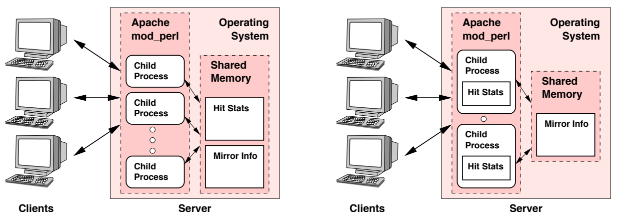

Figure 8. Consistent (left) and fast (right) operation modes of the module.

Our module replicates files by the means of point-to-point HTTP POST requests sent to a script installed on each of the servers. Optionally, content replication can be performed using SSL to ensure proper authentication of the servers, as only trusted servers should be able to store content by one another. The module can be configured to intercept all the requests received by a server or only those targeted at specific resources. Further, one can specify a minimum size for the resources that our module will watch and replicate, as well as their type (e.g., only HTML content and images). Finally, the module can be configured to only replicate individual resources, optionally rewriting some of the relative links, or to also replicate embedded resources (e.g., images). All of the module options, including the thresholds, are specified declaratively in a configuration file.

A major limitation of our approach is that it is only applicable to static content. Note, however, that "rich content" (e.g., images, movies) in the Web is mostly static and represents a major portion of the requests and data volume. Transparent replication of dynamic pages that use server-side scripting or back-end databases is a challenging endeavor, given that there is no a priori knowledge of which content is at risk of being hit by a flash crowd. Various techniques to support spikes of traffic in the context of dynamic content have been proposed in [11, 5, 26, 16, 8].

6. Experimental Evaluation

We have conducted an experimental evaluation of our prototype implementation in order to assess its performance and effectiveness at sustaining flash crowds. This section presents the results of our experimental study.

6.1. Experimental Setup

We have deployed our proactive hot spot avoidance module on a 1.5 GHz Intel Pentium IV machine with 512 MB of main memory running Linux 2.4.18. We used Apache 2.0.48 as Web server with Perl 5.8.0 and mod_perl 1.99.08. This configuration ensures good processing performance of the primary source, with the likely bottleneck being the network bandwidth. For the mirror servers, which have lower performance expectations, we used a set of 15 identical Sun Ultra 10 workstation with a 440 MHz processor and 256 MB of memory, running Solaris 2.8 and the Apache 2.0.47 Web server. All machines were part of the same switched network, with 100 Mb/s (full duplex) bandwidth capacity and under normal traffic conditions (note that, in practice, the maximum throughput of an Ethernet network is typically no more than 90% of the rated maximum). To simulate a typical distributed environment with the moderate-cost Web servers we are targeting with our hot spot avoidance mechanisms, we have limited the server bandwidth to 10 Mb/s (half duplex) by configuring the switch appropriately. This setup is adequate to run stress tests on the server and assess its scalability and performance, but it does not model the "user experience" of clients over wide area networks: end users will experience additional delays that depend on their connectivity and distance from the source and mirror servers.

To measure the performance of the Web server, we have primarily used the *httperf* [17] and *Autobench* [4] tools, which allow us to generate heavy HTTP workloads and simulate simultaneous connections from large client populations. To observe the impact of our module as a function of time during the flash crowd, we have used a modified version of the *siege* [21] application. The clients were running on a separate machine configured identically to the primary source server.

We have generated realistic data workload on the basis of the analysis of Section 3. The size of the HTML content most affected by the flash crowd was approximately 10 KB, with a 50 KB image embedded in the page (we ignored other embedded resources of less than 1 KB, such as logos, which do not significantly impact bandwidth usage). Further, the HTML page links to larger resources, such as a 400 KB brochure, which were also often requested during the flash crowd. We have therefore generated three representative data workloads with documents of size 10 KB, 50 KB, and 100 KB respectively. To better study the impact of the size, we have accessed the same resource repeatedly for each of the experiments, without mixing resources of different sizes. This corresponds to a best-case scenario for the Web server when not using our module, as the requested page is likely to be cached in memory.

6.2. Raw Server Performance

We have first observed the performance of the Web server under high request load. To that end, we have configured *httperf* to create $20,000$ connections, with 10 requests per connection and a number of new connections created per second varying between 10 and 150; this correspond to a maximum demanded rate of $1,500$ requests per second. We have set the response timeout to 5 seconds, which means that the lack of any server activity on the TCP connection for this duration will be considered to be an error, i.e., a performance failure (note that, due to TCP control traffic, this does not prevent the client from successfully receiving a reply after the timeout without reporting an error). All numbers are averages computed by *httperf* over all connections.

Figure 9 (a) shows the effective response rate achieved by the Web server for documents of various sizes, and Figure 9 (b) shows the corresponding network throughput (computed as the number of bytes sent and received on the TCP connections, i.e., without accounting for network headers or possible TCP retransmissions). The figures show results when the module is disabled and all files are served by the source server (10 KB, 50 KB, 100 KB), as well as when the module is active and only redirects the clients toward the mirrors (REDIRECT). In the latter case, we pre-enabled redirection mode ($T_R = \infty$) and we only measured the traffic at the server, i.e., without any subsequent request that regular clients would send to the mirror servers. Obviously, the performance of redirection does not depend on the size of the requested resource (we used the 100 KB file for the experiments).

We observe that the server can sustain the requested rate only up to a certain limit, which directly depends on the size of the requested document: $10, 20$, and 90 requests per second for documents of 10 KB, 50 KB, 100 KB respectively. Figure 9 (b) clearly shows that this limit is reached when the network becomes saturated around $1,000$ KB/s, which corresponds to approximately 8 Mb/s. One can note that the network throughput decreases after the saturation point for large documents; this can be explained by the large number of TCP retransmissions that are not accounted for in the statistics.

For the tests with the hot spot avoidance module, we observe a maximum effective rate of approximately 900 requests per second with a bandwidth that never exceeds 700 KB/s (less than 6 Mb/s). In this configuration, the processing capacity of the server becomes saturated before its network bandwidth because of the runtime overhead of the non-trivial computations performed by our module. Yet, the effective response rate is high enough to easily sustain a flash crowd like that analyzed in Section 3, with a peak rate of 215 requests during the busiest second, given that it does not depend on the size of the requested content. Clearly, redirections allow us to scale to much larger client populations than when directly serving the requested content. As we shall discuss next, clients also experience fewer performance failures because the server reaches its saturation point at higher request rates.

6.3. Performance Failures

When a Web server becomes saturated, it suffers from performance failures and clients experience long response delays or loss of service. To determine the impact of this problem, we have measured the response time and the number of timeout errors as a function of the demanded request rate. Note that, as clients are located nearby the server, the response time is an indication of the additional delay that a remote client will experience on top of the usual request latency (without taking into account the redirected request, if any).

We observe in Figures 9 (c) and 9 (d) that the server scales up to 900 requests per second with redirections before clients start experiencing errors. In contrast, when directly serving the documents, the response time becomes significant after only 90 requests per second for documents of 10 KB—earlier for larger documents—and a large proportion of the clients are not served in time (within 5 seconds) or are not served at all. Thus, in addition to increasing the success rate, redirections can also improve the response time experienced by clients under high load, even when taking into account the extra indirection to the mirror server.

6.4. Latency

We have finally studied the evolution of the response time during the occurrence of a flash crowd. To that end, we have simulated the arrival of an increasing number of clients that continuously request the same 5 resources from the server. We have used files of various sizes between $2,639$ and $51,100$ bytes corresponding to the largest resources (HTML page and images) among those requested more than $25,000$ times on the day of the flashcrowd. Clients wait for a random delay up to 1 second between successive requests, and a new client arrives every 2 seconds until 100 concurrent connections have been established. Clients terminate after they have issued 300 requests. We ran experiments using the consistent operation mode of the Web module and a random mirror selection strategy, with $b = 10$, $k = 1,000$, $T_C = 1$, $T_R = 0.5$, and $T_S = 10$. We deployed 8 mirror server, so that each of them receives a small fraction of the original traffic and never becomes network- or CPU-saturated. As all computers were running in the same intranet, network latency is negligible under normal operation and saturation of the source server is exacerbated upon flash crowd.

Figure 9 (e) shows the evolution of the response time experienced by the clients when the source directly serves the

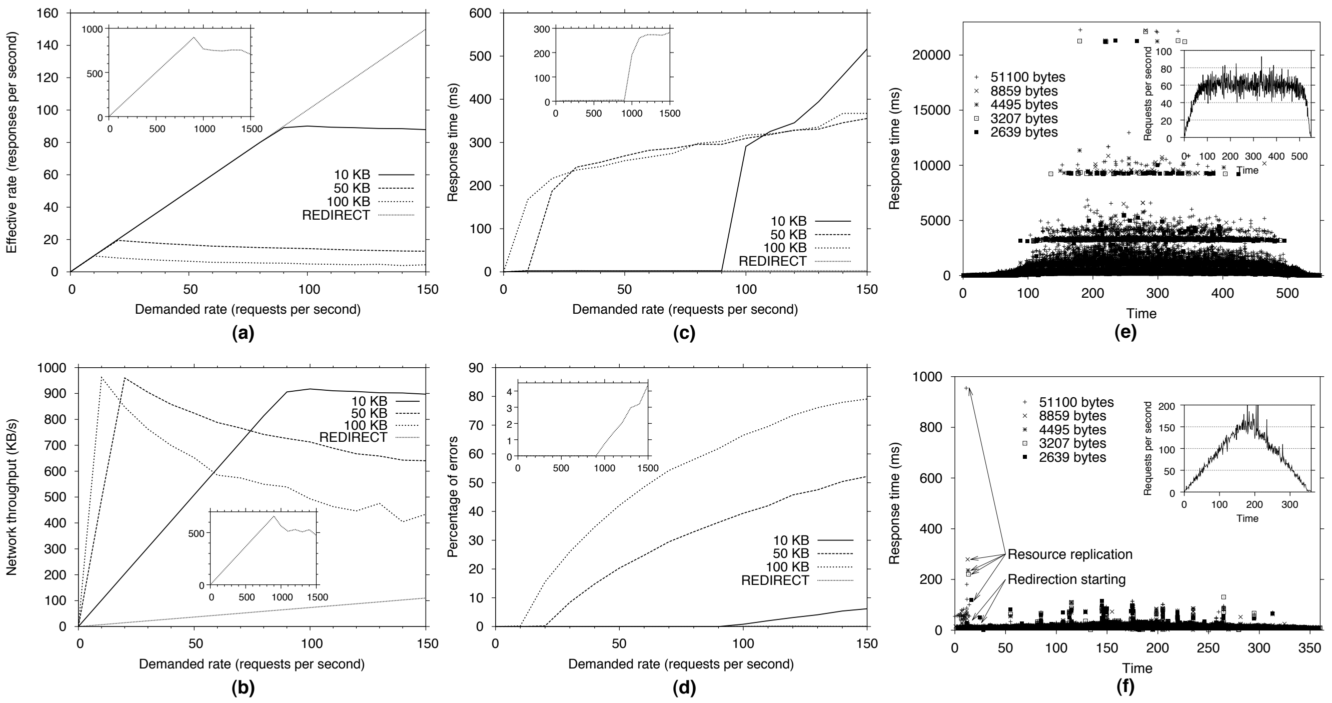

Figure 9. Left and center: response rate (a), network throughput (b), response time (c), and error rate (d) as a function of the demanded request rate (inner graphs zoom out on redirection mode). Right: response time experienced by clients during flash crowd *without* (e) and *with* (f) redirections.

requested resources. We observe that the latency increases steadily with the number of clients, which experience delays of more than 20 seconds at the peak of the flash crowd. We clearly see dense groups of responses around 3, 9, and 22 seconds that correspond to TCP retransmissions. Further, the throughput never exceeds 80 requests per second, i.e., less than half the request rate expected from a non-saturated server under the same client workload.

Figure 9 (f) shows the results of the same experiment when using the hot spot avoidance module (note that the measured response time includes both the request to the source server and the subsequent request to the mirror). We initially observe some contention at the source server as the number of concurrent clients increases. After 11 to 16 seconds, the resources are copied to the mirrors. This event appears clearly in Figure 9 (f), as resource replication is performed while processing a client request and delays the reply until after the copying process has completed (unsurprisingly, the replication of the largest file is the most time-consuming). Redirection is activated after 14 to 21 seconds and the response time remains generally small (except for minor spikes resulting from the uneven and fluctuating load of the servers) despite the high traffic and the delay introduced by the additional requests to the mirror servers. At the peak of the flash crowd, the throughput reaches 200 requests per second, which demonstrates that redirections allow us to serve more clients with a better quality of service.

7. Conclusion

In this paper, we have discussed the problem of flash crowds and their negative impact on Web servers, which may quickly become saturated by request traffic and suffer from various kind of failures, mostly performance and crash failures. We have analyzed the flash crowd phenomenon by studying a real log from a Web server that has been hit by a sudden surge of traffic immediately after being linked on a highly popular Web site. We have observed huge spikes in the request traffic and data volume at the occurrence of the flash crowd, and we have noticed that only a small set of resources has been requested repeatedly.

On the basis of this analysis, we have proposed a proactive hot spot avoidance mechanism implemented as a software module for the Apache Web server. The module is installed downstream from the servers in order to detect unexpected bursts of traffic. When this happens, the overloaded server performs on-the-fly replication of the affected content to peer servers, and subsequently redirects clients to the copies. This approach is “proactive” in the sense that the content replication is performed when detecting a traffic surge and anticipating a flash crowd. Using such techniques, groups of small- to medium-sized servers can team up and capitalize their common bandwidth to support temporary high traffic load. Experimental evaluation has demonstrated that our mechanisms can significantly increase the scalability of Web servers and that hot spots can effectively be avoided at low cost without need to upgrade the servers.

Besides extending the support of our mechanisms to some types of dynamic content, we are exploring the issue of locality-driven server organization and selection, in order to mirror content close to the clients and redirect them to nearby copies. To that end, we envision to use techniques similar to those put forth in the TOPLUS topology-aware peer-to-peer substrate [10], which organizes peers based on efficient proximity metrics derived from IP prefixes.

Acknowledgments: We are extremely grateful to Sean Adams for kindly providing us with the anonymized Web logs of the flash crowd analyzed in this paper.

References

[1] S. Adler. The Slashdot Effect: An analysis of Three Internet Publications. `http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html`.

[2] Akamai. `http://www.akamai.com`.

[3] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long. Managing Flash Crowds on the Internet. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling (MASCOTS)*, Oct. 2003.

[4] Autobench. `http://www.xenoclast.org/autobench/`.

[5] A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW)*, Aug. 2002.

[6] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. World Wide Web Caching: The Application-Level View of the Internet. *IEEE Communications*, 35(6):170–178, 1997.

[7] X. Chen and J. Heidemann. Experimental Evaluation of an Adaptive Flash Crowd Protection System. Technical Report ISI-TR-2003-573, USC/ISI, July 2003.

[8] K. Coleman, J. Norris, G. Candea, and A. Fox. Oncall: Defeating spikes with a free-market server cluster. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, May 2004.

[9] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.

[10] L. Garces-Erice, K. Ross, E. Biersack, P. Felber, and G. Urvoy-Keller. Topology-centric look-up service. In *Proceedings of the Fifth International Workshop on Networked Group Communications (NGC)*, Sept. 2003.

[11] G. Goldszmidt, K. Appleby, S. Fakhouri, L. Fong, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Océano – SLA based management of a computing utility. In *7th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2001.

[12] Google. `http://www.google.com`.

[13] J. Hu and D. Schmidt. JAWS: A Framework for High-performance Web Servers. In M. Fayad and R. Johnson, editors, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. John-Wiley, 1999.

[14] S. Iyer, A. Rowstron, and P. Druschel. SQUIRREL: A Decentralized, Peer-to-Peer Web Cache. In *Principles of Distributed Computing (PODC)*, July 2002.

[15] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of International World Wide Web Conference (WWW)*, May 2002.

[16] E. Lassettre, D. Coleman, Y. Diao, S. Froehlich, J. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye. Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, Oct. 2003.

[17] D. Mosberger and T. Jin. httperf—A Tool for Measuring Web Server Performance. Technical Report HPL-98-61, Hewlett-Packard Laboratories, Mar. 1998.

[18] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[19] G. Pierre, I. Kuz, and M. van Steen. Adaptive Replicated Web Documents. Technical Report IR-477, Vrije Universiteit, Amsterdam, Sept. 2000.

[20] G. Pierre and M. van Steen. Globule: a Platform for Self-Replicating Web Documents. In *Proceedings of the 6th International Conference on Protocols for Multimedia Systems*, Oct. 2001.

[21] Siege. `http://www.joedog.org/siege/`.

[22] Squid. `http://www.squid-cache.org`.

[23] T. Stading, P. Maniatis, and M. Baker. Peer-to-Peer Caching Schemes to Address Flash Crowds. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.

[24] A. Stavrou, D. Rubenstein, and S. Sahu. A Lightweight, Robust P2P System to Handle Flash Crowds. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP)*, Nov. 2002.

[25] Thttpd. `http://www.acme.com/software/thttpd/`.

[26] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, Oct. 2001.

[28] W. Zhao and H. Schulzrinne. Predicting the Upper Bound of Web Traffic Volume Using a Multiple Time Scale Approach. In *Proceedings of International World Wide Web Conference (WWW)*, May 2003.