DotSlash: A Self-configuring and Scalable Rescue System for Handling Web Hotspots Effectively*

Weibin Zhao and Henning Schulzrinne

Columbia University, New York NY 10027, USA
{zwb,hgs}@cs.columbia.edu

Abstract. DotSlash allows different web sites to form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site. As a rescue system, DotSlash intervenes when a web site becomes heavily loaded, and is phased out once the workload returns to normal. It aims to complement the existing web server infrastructure to handle short-term load spikes effectively. DotSlash is self-configuring, scalable, cost-effective, easy to use, and transparent to clients. It targets small web sites, although large web sites can also benefit from it. We have implemented a prototype of DotSlash on top of Apache. Experiments show that using DotSlash a web server can increase the request rate it supported and the data rate it delivered to clients by an order of magnitude, even if only HTTP redirect is used. Parts of this work may be applicable to other services such as Grid computational services.

1 Introduction

As more web sites experience a request load that can no longer be handled by a single server, using multiple servers to serve a single site becomes a widespread approach. Traditionally, a distributed web server system has used a fixed number of dedicated servers based on capacity planning, which works well if the request load is relatively consistent and matches the planned capacity. However, web requests could be very bursty. A well-identified problem web hotspots (also known as flash crowds or the Slashdot effect [2]) may trigger a large load increase but only last for a short time [14, 24]. For such situations, overprovisioning a web site is not only uneconomical but also difficult since the peak load is hard to predict [16].

To handle web hotspots effectively, we advocate dynamic allocation of server capacity from a server pool distributed globally because the access link of a local network could become a bottleneck. As an example of global server pools, content delivery networks (CDNs) [27] have been used by large web sites, but small web sites often cannot afford the cost particularly since they may need these services very rarely. We seek a more cost-effective mechanism. As different web sites (e.g., different types or in different locations) are less likely to experience their peak request loads at the same time, they could form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site [10]. Based on this observation, we designed *DotSlash* which allows a web site to build an adaptive distributed

* This work was supported in part by the National Science Foundation (ANI-0117738).

web server system on the fly to expand its capacity by utilizing spare capacity at other sites. Using DotSlash, a web site not only has a fixed set of *origin servers*, but also has a changing set of *rescue servers* drafted from other sites. A web server allocates and releases rescue servers based on its load conditions. The rescue process is completely self-managing and transparent to clients.

DotSlash does not aim to support a request load that is persistently higher than a web site's planned capacity, but rather to complement the existing web server infrastructure to handle short-term load spikes effectively. We envision a spectrum of mechanisms for web sites to handle load spikes. Infrastructure-based approaches should handle the request load sufficiently in most cases (e.g., 99.9% of time), but they might be too expensive for short-term enormous load spikes and insufficient for unexpected load increases. For these cases, DotSlash intervenes so that a web site can support its request load in more cases (e.g., 99.999% of time). In parallel, a web site can use service degradation [1] such as turning off dynamic content and serving a trimmed version of static content under heavily-loaded conditions. As the last resort, a web site can use admission control [31] to reject a fraction of requests and only admit preferred clients.

DotSlash has the following advantages. First, it is self-configuring in that service discovery [13] is used to allow servers of different web sites to learn about each other dynamically, rescue actions are triggered automatically based on load conditions, and a rescue server can serve the content of its origin servers on the fly without the need of any advance configuration. Second, it is scalable because a web server can expand its capacity as needed by using more rescue servers. Third, it is very cost-effective since it utilizes spare capacity in a web server community to benefit any participating server, and it is built on top of the existing web server infrastructure, without incurring any additional hardware cost. Fourth, it is easy to use because standard DNS mechanisms and HTTP redirect are used to offload client requests from an origin server to its rescue servers, without the need of changing operating system or DNS server software. An add-on module to the web server software is sufficient to support all needed functions. Fifth, it is transparent to clients since it only uses server-side mechanisms. Client browsers remain unchanged, and client bookmarks continue to work. Finally, an origin server has full control of its own rescue procedure, such as how to choose rescue servers and when to offload client requests to rescue servers.

DotSlash targets small web sites, although large web site can also benefit from it. We focus on load migration for static web pages in this paper, and plan to investigate load migration for dynamic content in the next stage of this project. Parts of this work may be applicable to other services such as Grid computational services [12]. The remainder of this paper is organized as follows. We discuss related work in Section 2, give an overview of DotSlash in Section 3, present DotSlash design, implementation and evaluation in Section 4, 5 and 6, respectively, and conclude in Section 7.

2 Related Work

Caching [29] provides many benefits for web content retrieval, such as reducing bandwidth consumption and client-perceived latency. Caching may appear at several different places, such as client-side proxy caching, intermediate network caching, and server-side reverse caching, many of which are not controlled by origin web servers. DotSlash uses caching at rescue servers to relieve the load spike at an origin server, where caching is set up on demand and fully controlled by the origin server.

CDN [27] services deliver part or all of the content for a web site to improve the performance of content delivery. As an infrastructure-based approach, CDN services are good for reinforcing a web site in a long run, but less efficient for handling short-term load spikes. Also, using CDN services needs advance configurations such as contracting with a CDN provider and changing the URIs of offloading objects (e.g., Akamaized [3]). As an alternative mechanism to CDN services, DotSlash offers cost-effective and automated rescue services for better handling short-term load spikes.

Distributed web server systems are a widespread approach to support high request loads and reduce client-perceived delays. These systems often use replicated web servers (e.g., ScalaServer [5] and GeoWeb [9]), with a focus on load balancing and serving a client request from the closest server. In contrast, DotSlash allows an origin server to build a distributed system of heterogeneous rescue servers on demand so as to relieve the heavily-loaded origin server. DC-Apache [17] supports collaborations among heterogeneous web servers. However, it relies on static configuration to form collaborating server groups, which limits its scalability and adaptivity to changing environments. Also, DC-Apache incurs a cost for each request by generating all hyperlinks dynamically. DotSlash addresses these issues by forming collaborating server groups dynamically, and using simpler and widely applicable mechanisms to offload client requests. Backslash [25] suggests using peer-to-peer (P2P) overlay networks to build distributed web server systems and using distributed hash table to locate resources.

The Internet Engineering Task Force (IETF) has developed a model for content internetworking (CDI) [11, 23]. The DotSlash architecture appears to be a special case of the CDI architecture, where each web server itself is a content network. However, the CDI framework does not address the issue of using dynamic server allocation and dynamic rate adjustment based on feedback to handle short-term load spikes, which is the main focus of DotSlash.

Client-side mechanisms allow clients to help each other so as to alleviate server-side congestion and reduce client-perceived delays. An origin web server can mediate client cooperation by redirecting a client to another client that has recently downloaded the URI, e.g., Pseudoserving [15] and CoopNet [20]. Clients can also form P2P overlay networks and use search mechanisms to locate resources, e.g., PROOFS [26] and BitTorrent [7]. Client-side P2P overlay networks have advantages in sharing large and popular files, which can reduce request loads at origin web servers. In general, client-side mechanisms scale well as the number of clients increases, but they are not transparent to clients, which are likely to prevent widespread deployment.

Grid technologies allow "coordinated resource sharing and problem solving in dynamic, multi-institutional organizations" [12], with a focus on large-scale computational problems and complex applications. The sharing in Grid is broader than simply file exchange; it can involve direct access to computers, software, data, and other resources. In contrast, DotSlash employs inter-web-site collaborations to handle web hotspots effectively, with an emphasis on overload control at web servers and disseminating popular files to a large number of clients.

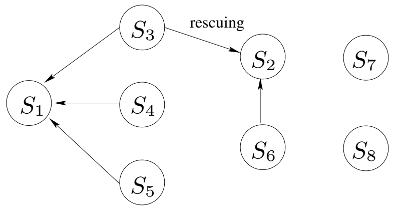


Fig. 1. An example for DotSlash rescue relationships

3 DotSlash Overview

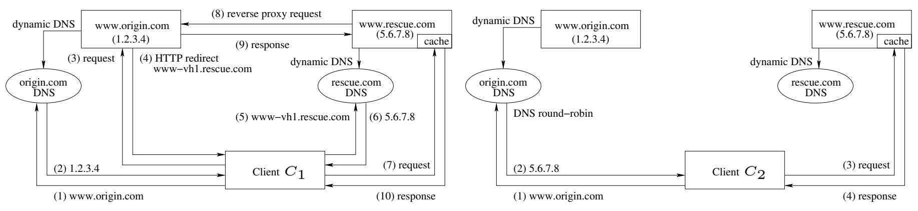
DotSlash uses a mutual-aid rescue model. A web server joins a mutual-aid community by registering itself with a DotSlash service registry, and contributing its spare capacity to the community. In case of being heavily loaded, a participating server discovers and uses spare capacities at other servers in its community via DotSlash rescue services. In our current prototype, DotSlash is intended for a cooperative environment, and thus no payment is involved in obtaining rescue services.

In DotSlash, a web server is in one of the following states at any time: *SOS state* if it gets rescue services from others, *rescue state* if it provides rescue services to others, and *normal state* otherwise. These three states are mutually exclusive: a server is not allowed to get a rescue service as well as to provide a rescue service at the same time. Using this rule can avoid complex rescue scenarios (e.g., a rescue loop where S_1 requests a rescue service from S_2, S_2 requests a rescue service from S_3, and S_3 requests a rescue service from S_1), and keep DotSlash simple and robust without compromising scalability. Throughout this paper, we use the notation origin server and rescue server in the following way. When two servers set up a rescue relationship, the one that benefits from the rescue service is the origin server, and the one that provides the rescue service is the rescue server. Fig. 1 shows an example of rescue relationships for eight web servers, where an arrow from S_y to S_x denotes that S_y provides a rescue service to S_x. In this figure, S_1 and S_2 are origin servers, S_3, S_4, S_5 and S_6 are rescue servers, and S_7 and S_8 have not involved themselves with rescue services.

3.1 Rescue Examples

In DotSlash, an origin server uses HTTP redirect and DNS round robin to offload client requests to its rescue servers, and a rescue server serves as a reverse caching proxy for its origin servers. There are four rescue cases: (1) HTTP redirect (at the origin server) and cache miss (at the rescue server), (2) HTTP redirect and cache hit, (3) DNS round robin and cache miss, and (4) DNS round robin and cache hit. We show examples for case 1 and 4 next; case 2 and 3 can be derived similarly.

In Fig. 2, the origin server is *www.origin.com* with IP address *1.2.3.4* (referred to as S_o), and the rescue server is *www.rescue.com* with IP address *5.6.7.8* (referred to as S_r). S_r has assigned an alias *www-vh1.rescue.com* to S_o, and S_o has added S_r's IP address



(a) For HTTP redirect and cache miss (b) For DNS round robin and cache hit

Fig. 2. Rescue examples

to its round robin local DNS. Fig. 2(a) gives an example for case 1, where client C_1 follows a ten-step procedure to retrieve *http://www.origin.com/index.html*:

1. C_1 resolves S_o's domain name *www.origin.com*;
2. C_1 gets S_o's IP address *1.2.3.4*;
3. C_1 makes an HTTP request to S_o using *http://www.origin.com/index.html*;
4. C_1 gets an HTTP redirect from S_o as *http://www-vh1.rescue.com/index.html*;
5. C_1 resolves S_r's alias *www-vh1.rescue.com*;
6. C_1 gets S_r's IP address *5.6.7.8*;
7. C_1 makes an HTTP request to S_r using *http://www-vh1.rescue.com/index.html*;
8. S_r makes a reverse proxy request to S_o using *http://www.origin.com/index.html* because of a cache miss for *http://www-vh1.rescue.com/index.html*;
9. S_o sends the requested file to S_r;
10. S_r caches the requested file, and returns the file to C_1.

Fig. 2(b) gives an example for case 4, where client C_2 follows a four-step procedure to retrieve *http://www.origin.com/index.html*:

1. C_2 resolves S_o's domain name *www.origin.com*;
2. C_2 gets S_r's IP address *5.6.7.8* due to DNS round robin at S_o's local DNS;
3. C_2 makes an HTTP request to S_r using *http://www.origin.com/index.html*;
4. C_2 gets the requested file from S_r because of a cache hit.

4 DotSlash Design

The main focus of DotSlash is to allow a web site to build an adaptive distributed web server system in a fully automated way. DotSlash consists of dynamic virtual hosting, request redirection, workload monitoring, rescue control, and service discovery.

4.1 Dynamic Virtual Hosting

Dynamic virtual hosting allows a rescue server to serve the content of its origin servers on the fly. Existing virtual hosting (e.g., Apache [4]) needs advance configurations: registering virtual host names in DNS, creating *DocumentRoot* directories, and adding directives to the configuration file to map virtual host names to *DocumentRoot* directories. DotSlash handles all these configurations dynamically.

A rescue server generates needed virtual host names dynamically by adding a sequence number component to its configured name, e.g., *host-vh<seqnum>.domain* for *host.domain*, where *<seqnum>* is monotonically increasing. Virtual host names are registered using *A* records via dynamic DNS updates [28]. We have set up a domain *dot-slash.net* that accepts virtual host name registrations. For example, *www.rescue.com* can obtain a unique host name *foo* in *dot-slash.net*, and register its virtual host names as *foo-vh<seqnum>.dot-slash.net*. A rescue server assigns a unique virtual host name to each of its origin servers, which is used in the HTTP redirects issued from the corresponding origin server.

As a rescue server, *www.rescue.com* may receive requests using three different kinds of *Host* header fields: its configured name *www.rescue.com*, an assigned virtual host name such as *www-vh1.rescue.com*, or an origin server name such as *www.origin.com*. Its own content is requested in the first case, whereas the content of its origin servers is requested in the last two cases. Moreover, the second case is due to HTTP redirects, and the third case is due to DNS round robin. A rescue server maintains a table to map assigned virtual host names to its origin servers. To map the *Host* header field of a request, a rescue server checks both the virtual host name and the origin server name in each mapping entry; if either one matches, the origin server name is returned. Due to client-side caching, web clients may continue to request an origin server's content from its old rescue servers. To handle this situation properly, a rescue server does not remove a mapping entry immediately after the rescue service has been terminated, but rather keeps the mapping entry for a configured time such as 24 hours, and redirects such a request back to the corresponding origin server via an HTTP redirect.

A rescue server works as a reverse caching proxy for its origin servers. For example, when *www.rescue.com* has a cache miss for *http://www-vh1.rescue.com/index.html*, it maps *www-vh1.rescue.com* to *www.origin.com*, and issues a reverse proxy request for *http://www.origin.com/index.html*. Using reverse caching proxy offers a few advantages. First, as files are replicated on demand, the origin server incurs low cost since it does not need to maintain states for replicated files and can avoid transferring files that are not requested at the rescue server. Second, as proxy and caching are functions supported by most web server software, it is simple to use reverse proxying to get needed files, and use the same caching mechanisms to cache proxied files and local files.

4.2 Request Redirection

Request redirection [8, 6, 30] allows an origin server to offload client requests to its rescue servers, which involves two aspects: the mechanisms to offload client requests and the policies to choose a rescue server among multiple choices. A client request can be redirected by the origin server's authoritative DNS, the origin server itself, or a redirector at transport layer (content-blind) or application layer (content-aware). Redirection policies can be based on load at rescue servers, locality of requested files at rescue servers, and proximity between the client and rescue servers.

DotSlash uses two mechanisms for request redirections: DNS round robin at the first level for crude load distribution, and HTTP redirect at the second level for finegrained load balancing. DNS round robin can reduce the request arrival rate at the origin server, and HTTP redirect can increase the service rate of the origin server because an HTTP redirect is much cheaper to serve than the original content. Both mechanisms can increase the origin server's throughput for request handling.

We investigated three options for constructing redirect URIs: IP address, virtual directory, and virtual host name. Using the rescue server's IP address can save the client's DNS lookup time for the rescue server's name, but the rescue server is unable to tell whether a request is for itself or for one of its origin servers. Using a virtual directory such as */dotslash-vh*, *http://www.origin.com/index.html* can be redirected as *http://www.rescue.com/dotslash-vh/www.origin.com/index.html*. The problem is that it does not work for embedded relative URIs. DotSlash uses virtual host names, which allows proper virtual hosting at the rescue server, and works for embedded relative URIs.

In terms of redirection policies, DotSlash uses standard DNS round robin without modifying the DNS server software, and uses weighted round robin (WRR) for HTTP redirects, where the weight is the allowed redirect data rate assigned by each rescue server. Due to factors such as caching and embedded relative URIs, the redirect data rate seen by the origin server may be different from that served by the rescue server. For simplicity, an origin server only controls the data rate of redirected files, not including embedded objects such as images, and relies on a rate feedback from the rescue server to adjust its redirect data rate (see Section 4.4 for details).

Redirection needs to be avoided for communications between two collaborating servers and for requests of getting server status information. On one hand, a request sender (a web client or a web server) needs to bypass DNS round robin by using the server's IP address directly in the following cases: when a server initiates a rescue connection to another server, when a rescue server makes a reverse proxy request to its origin server, and when a client retrieves a server's status information. On the other hand, a request receiver (i.e., a web server) needs to avoid performing an HTTP redirect if the request is from a rescue server, or if the request is for the server's status information.

4.3 Workload Monitoring

Workload monitoring allows a web server to react quickly to load changes. Major DotSlash parameters are summarized in Table 1. We measure the utilization of each resource at a web server separately. According to a recent study [20], network bandwidth is the most constrained resource for most web sites during hotspots. We focus on monitoring network utilization ρ_n in this paper. We use two configurable parameters, lower threshold ρ_n^l and upper threshold ρ_n^u, to define three regions for ρ_n: lightly loaded region $[0, \rho_n^l)$, desired load region $[\rho_n^l, \rho_n^u]$, and heavily loaded region $(\rho_n^u, 100\%]$. Furthermore, we define a reference utilization $\hat{\rho_n}$ as $(\rho_n^l + \rho_n^u)/2$.

In DotSlash, we monitor outbound HTTP traffic within a web server, without relying on an external module to monitor traffic on the link. We assume there is no significant other traffic besides HTTP at a web server, and assume a web server has a symmetric link or its inbound bandwidth is greater than its outbound bandwidth, which is true, for example, for a web server behind DSL. Since a web server's outbound data rate

Table 1. Major DotSlash parameters, where type C is for configurable parameters, type O is for measured outputs, type I is for control inputs, and type D is for derived parameters

Parameter	Description	Type
ρ_n^l and ρ_n^u	lower and upper threshold for network utilization, default 50% and 75%	C
λ_d^m	maximum data rate (kB/s) for outbound HTTP traffic	C
τ	control interval, default 1 second	C
α	used in exponentially weighted moving average filter, default 0.5	C
λ_d	real data rate (kB/s) of outbound HTTP traffic	O
λ_{rd}	real redirect data rate (kB/s)	O
λ_{rd}^a	allowed redirect data rate (kB/s)	I
P_r	redirect probability	I
ρ_n	network utilization, $\rho_n = \lambda_d/\lambda_d^m$	D
$\hat{\rho_n}$	reference network utilization, $\hat{\rho_n} = (\rho_n^u + \rho_n^l)/2$	D
$\hat{\lambda_d}$	reference data rate (kB/s), $\hat{\lambda_d} = \hat{\rho_n}\lambda_d^m$	D
β	adjustment factor for control inputs, $\beta = \rho_n/\hat{\rho_n}$	D

is normally greater than its inbound data rate, it should be sufficient to only monitor outbound HTTP traffic.

Due to header overhead (such as TCP and IP headers) and retransmissions, the HTTP traffic rate monitored by DotSlash is less than the real traffic rate on the link. Since the header overhead is relatively constant and other overheads are usually small, to simplify calculation, we use a configurable parameter λ_d^m to denote the maximum data rate for outbound HTTP traffic, where $\lambda_d^m = BU$, B is the network bandwidth, and U is the percentage of bandwidth that is usable for HTTP traffic. We perform a special accounting for HTTP redirects because they may account for a large percentage of HTTP responses and their header overhead is large compared to their small sizes. For an HTTP redirect response of n bytes, its accounting size $A_r = (n + O)U$ bytes, where O is the header overhead. A web server sends five TCP packets for each HTTP redirect: one for establishing the TCP connection, one for acknowledging the HTTP request, one for sending the HTTP response, and two for terminating the TCP connection. The first TCP header (SYN ACK) is 40 bytes, and the rest four TCP headers are 32 bytes each. Thus, $O = (40 + 32 * 4) + 20 * 5 + (14 + 4) * 5 = 358$ bytes, which includes the TCP and IP headers, and the Ethernet headers and trailers.

4.4 Rescue Control

Rescue control allows a web server to tune its resource utilization by using rescue actions that are triggered automatically based on load conditions. To control ρ_n within the desired load region $[\rho_n^l, \rho_n^u]$, overload control actions are triggered if $\rho_n > \rho_n^u$, and under-load control actions are triggered if $\rho_n < \rho_n^l$. To control the utilization of multiple resources, overload control actions are triggered if *any* resource is heavily loaded, and under-load control actions are triggered if *all* resources are lightly loaded.

Origin servers and rescue servers use different control parameters. An origin server controls the redirect probability P_r by increasing P_r if $\rho_n > \rho_n^u$ and decreasing P_r if

$\rho_n < \rho_n^l$, whereas a rescue server controls the allowed redirect data rate λ_{rd}^a for each of its origin servers by decreasing λ_{rd}^a if $\rho_n > \rho_n^u$ and increasing λ_{rd}^a if $\rho_n < \rho_n^l$. An origin server should ensure the real redirect data rate $\lambda_{rd} \leq \lambda_{rd}^a$, but a rescue server may experience $\lambda_{rd} > \lambda_{rd}^a$.

We use the following control strategies. A configurable parameter τ denotes the control interval, which is the smallest time unit for performing workload monitoring and rescue control. Other time intervals are specified as a multiple of the control interval. To handle stochastics, we apply an exponentially weighted moving average filter to ρ_n, P_r and λ_{rd}^a. Using ρ_n as an example, $\overline{\rho_n(k)} = \alpha\overline{\rho_n(k-1)} + (1-\alpha)\rho_n(k)$, where $\rho_n(k)$ is the current raw measurement, $\overline{\rho_n(k)}$ is the filtered value of $\rho_n(k)$, $\overline{\rho_n(k-1)}$ is the previous filtered value, and α is a configurable parameter with a default value 0.5. If multiple rescue server candidates are available, the one with the largest rescue capacity should be used first. This policy can help an origin server to keep the number of its rescue servers as small as possible. Minimizing the number of rescue servers can reduce their cache misses, and thus reduce the data transfers at the origin server.

The DotSlash rescue protocol (DSRP) is an application-level request-response protocol using single-line pure text messages. A request has a command string (starting with a letter) followed by optional parameters, whereas a response has a response code (three digits) followed by the response string and optional parameters. DSRP defines three requests: *SOS* for initiating a rescue relationship, *RATE* for adjusting a redirect data rate, and *SHUTDOWN* for terminating a rescue relationship. An *SOS* request is always sent by an origin server, and a *RATE* request is always sent by a rescue server, but a *SHUTDOWN* request may be sent by an origin server or a rescue server. To initiate a rescue relationship, an origin server sends an *SOS* request to a chosen rescue server candidate. The request has the following parameters: the origin server's fully qualified domain name, its IP address, and its port number for web requests. When a web server receives an *SOS* request, it can accept the request by sending a "200 OK" response or reject the request by sending a "403 Reject" response. A "200 OK" response has the following parameters: a unique alias of the rescue server assigned to the origin server, the rescue server's IP address, the rescue server's port number for web requests, and the allowed redirect data rate that the origin server can offload to the rescue server.

Fig. 3 summarizes DotSlash rescue actions and state transitions. We describe rescue actions in each state next. The normal state has two rescue actions: initial allocation and initial rescue. For the first case, if a web server is heavily loaded (i.e., $\rho_n > \rho_n^u$), then it needs to allocate its first rescue server, set P_r to 0.5, and switch to the SOS state. For the second case, if a web server receives a rescue request and it is lightly loaded (i.e., $\rho_n < \rho_n^l$), then it can accept the rescue request, set λ_{rd}^a to $(\hat{\rho_n} - \rho_n)\lambda_d^m$ or a smaller value determined by a rate allocation policy, and switch to the rescue state.

The SOS state has four rescue actions: increase P_r, additional allocation, decrease P_r, and release. For the first case, if an origin server is heavily loaded and it has unused redirect capacity (i.e., $\lambda_{rd} < \lambda_{rd}^a$), then it needs to increase P_r until P_r reaches 1. For the second case, if an origin server is heavily loaded and it has run out of redirect capacity (i.e., λ_{rd} equals λ_{rd}^a), then it needs to allocate an additional rescue server so as to increase its redirect capacity. For the third case, if an origin server is lightly loaded and it still redirects requests to rescue servers (i.e., $P_r > 0$), then it needs to decrease

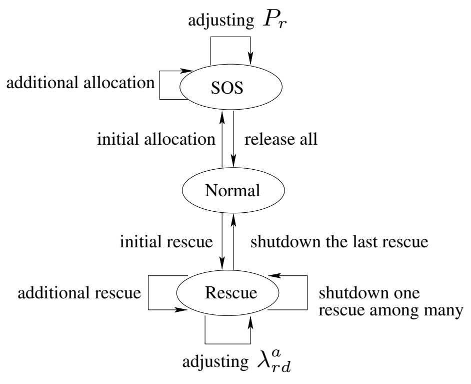


Fig. 3. DotSlash rescue actions and state transitions

P_r until P_r reaches 0. For the last case, if an origin server has been lightly loaded and has not redirected requests to rescue servers (i.e., P_r is 0) for a configured number of consecutive control intervals, then it needs to release all rescue servers. Fig. 4 gives the algorithm for adjusting P_r at an origin server, which increases P_r if $\rho_n > \rho_n^u$, and decreases P_r if $\rho_n < \rho_n^l$. The adjustment is controlled by parameter $\beta = \rho_n / \hat{\rho_n}$, where $\beta > 1$ for increase since $\rho_n > \rho_n^u > \hat{\rho_n}$, and $\beta < 1$ for decrease since $\rho_n < \rho_n^l < \hat{\rho_n}$. Further, the adjustment is smoothed by using an exponentially weighted moving average filter with $\alpha = 0.5$. To avoid infinite convergence, an increase from above 0.99 is set to 1, and a decrease from below 0.1 is set to 0. To react quickly to load spikes, an increase from below 0.5 is set to 0.5.

The rescue state has five rescue actions: decrease λ_{rd}^a, heavy-load shutdown, increase λ_{rd}^a, additional rescue, and idle shutdown. For the first case, if a rescue server is heavily loaded and its $\lambda_{rd}^a > 0$, then it needs to decrease λ_{rd}^a until λ_{rd}^a reaches 0. For the second case, if a rescue server is heavily loaded and its λ_{rd}^a is 0, then it needs to shutdown the rescue relationship. When a rescue server has shutdown all rescue relationships, it switches to the normal state. For the third case, when a rescue server is lightly loaded and $\lambda_{rd}^a < \hat{\lambda_d}$, then it can increase λ_{rd}^a. Note that a rescue server should not increase λ_{rd}^a if λ_{rd} is far below λ_{rd}^a. For the fourth case, if a rescue server is lightly loaded, and it receives a new rescue request, then it can accept the rescue request, and assign a λ_{rd}^a to the new origin server. By doing so, the rescue server will have multiple origin servers, and a separate λ_{rd}^a is assigned to each origin server. For the last case, if a rescue server has an origin server whose λ_{rd} has been 0 for a configured number of consecutive control intervals, then the rescue server should shutdown the rescue relationship so as to release rescue resources in case of the origin server failure or network separation. Fig. 5 gives the algorithm for adjusting λ_{rd}^a at a rescue server, which decreases λ_{rd}^a if $\rho_n > \rho_n^u$, and increases λ_{rd}^a if $\rho_n < \rho_n^l$. This algorithm is very similar to the algorithm shown in Fig. 4. However, these two algorithms make adjustments in opposite directions because their adjusting factors are β and $1/\beta$, respectively. We keep the adjusting factor for λ_{rd}^a within the range of $[0.5, 2]$ to avoid over-reacting adjustments. Also, we keep the filtered value of λ_{rd}^a as an integer.

```
// Compute $\beta$
$\beta = \rho_n / \hat{\rho_n}$;
```

```
// Increase $P_r$ if $\rho_n > \rho_n^u$
if ($P_r < 1$) {
  if ($P_r < 0.5$) {
    $P_r = 0.5$;
  } else if ($P_r > 0.99$) {
    $P_r = 1$;
  } else {
    $t = \mathsf{min}(\beta P_r, 1)$;
    $P_r = \alpha P_r + (1 - \alpha)t$;
  }
}
```

```
// Decrease $P_r$ if $\rho_n < \rho_n^l$
if ($P_r > 0$) {
  if ($P_r < 0.1$) {
    $P_r = 0$;
  } else {
    $t = \beta P_r$;
    $P_r = \alpha P_r + (1 - \alpha)t$;
  }
}
```

Fig. 4. Algorithm for adjusting P_r at an origin server

```
// Compute $\beta$
$\beta = \rho_n / \hat{\rho_n}$;
if ($\beta < 0.5$) {
  $\beta = 0.5$;
} else if ($\beta > 2$) {
  $\beta = 2$;
}
```

```
// Decrease $\lambda_{rd}^a$ if $\rho_n > \rho_n^u$
if ($\lambda_{rd}^a > 0$) {
  $t = \lambda_{rd}^a / \beta$;
  $\lambda_{rd}^a = (\mathsf{int})(\alpha \lambda_{rd}^a + (1 - \alpha)t)$;
}
```

```
// Increase $\lambda_{rd}^a$ if $\rho_n < \rho_n^l$
if ($\lambda_{rd}^a < \hat{\lambda_d}$) {
  $t = \mathsf{min}(\lambda_{rd}^a / \beta, \hat{\lambda_d})$;
  $\lambda_{rd}^a = (\mathsf{int})(\alpha \lambda_{rd}^a + (1 - \alpha)t)$;
}
```

Fig. 5. Algorithm for adjusting λ_{rd}^a at a rescue server

4.5 Service Discovery

Service discovery allows servers of different web sites to learn about each other dynamically and collaborate without any administrator intervention. DotSlash uses the Service Location Protocol (SLP) [13] since it is an IETF proposed standard for service discovery in IP networks, and it is flexible, lightweight and powerful. Based on the SLP mesh enhancement (mSLP [33]), DotSlash uses multiple well-known service registries that maintain a fully-meshed peer relationship. A web server can use any service registry to register its information and to search information about other web servers. Service registrations received by one registry will be propagated to other registries as soon as possible, and anti-entropy [32] is used to ensure consistency among all service registries. Only a small number of such service registries are needed for reliability and scalability. All of them serve the scope "DotSlash" (reserved for DotSlash rescue services) so that they will not affect local service discovery.

The template for DotSlash rescue services has the following attributes: the domain name for the web server, its IP address which is used to bypass DNS round robin, its port number for web requests, its port number for DotSlash rescue services, and the current allowed redirect data rate λ_{rd}^a computed as $\mathsf{max}((\hat{\rho_n} - \rho_n)\lambda_d^m, 0)$. A web server performs service registrations and searches periodically with a configurable interval τ_r and τ_s, respectively. To get ready for load spikes, a web server maintains a list of rescue server candidates. A DotSlash service search request uses preference filters [34] that allow the registry to sort the search result based on λ_{rd}^a and to only return the desired number of matching entries, which is useful if many entries match a search request.

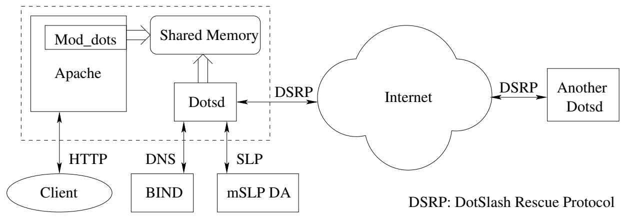


Fig. 6. DotSlash software architecture

5 Implementation

We use Apache [4] as our base system since it is open source and is the most popular web server [19]. Fig. 6 shows the DotSlash software architecture. DotSlash is implemented as two parts: *Mod_dots* and *Dotsd*. Mod_dots is an Apache module that supports DotSlash functions related to client request processing, including accounting for each response, HTTP redirect, and dynamic virtual hosting. Dotsd is a daemon that accomplishes other DotSlash functions, including service discovery, dynamic DNS updates, and rescue control and management. For convenience, Dotsd is started within the Apache server, and is shutdown when the Apache server is shutdown. Dotsd and Mod_dots share control data structures via shared memory. DNS servers and DotSlash service registries are DotSlash components external to the Apache server. We use BIND as DNS servers, and use mSLP Directory Agents (DAs) as DotSlash service registries. A web server interacts with other web servers via its Dotsd using DSRP carried by TCP.

The control data in shared memory are divided into two parts: a workload meter for the web server itself, and a peer table for collaborating web servers. The peer table maintains accounting information of redirected traffic for peers. Traffic accounting is performed in two time scales: the current control interval and the server's lifetime (from the server's starting time to now). The former accounting is used to trigger rescue actions, and the corresponding counters are reset to zero at the end of the current control interval. The latter accounting allows computing various average traffic rates by sampling the corresponding counters at desired time intervals.

Dotsd is implemented using pthread. It has two main threads: a control thread and a a DSRP server. The control thread runs at the end of each control interval for processing tasks that need to be done periodically such as computing the current workload level, triggering rescue actions if needed, and checking whether it needs to perform service discovery. The DSRP server accepts connections from other Dotsds and creates a new thread for processing each accepted connection. Dotsd also includes three clients: a DNS client for dynamic DNS updates, an SLP Service Agent for service registrations, and an SLP User Agent for service searches.

Mod_dots handles traffic accounting, performs HTTP redirects for an origin server, supports dynamic virtual hosting for a rescue server, and implements a content handler for */dotslash-status* so that a request for *http://host.domain/dotslash-status* can retrieve the current DotSlash status for the web server *host.domain*.

6 Evaluation

For a web server, we use two performance metrics D and R, where D is the maximum data rate of HTTP responses delivered to clients, and R is the maximum request rate supported. Our goal is to improve a web server's D and R by using DotSlash rescue services. For a web server without using DotSlash, its D and R can be estimated as λ_d^m and $\lambda_d^m/(F+H)$, respectively, where F is the average size of requested files, and H is the average HTTP header size of responses, assuming the CPU is not a bottleneck. For any web server, the maximum rate of HTTP redirects it can support can be estimated as λ_d^m/A_r, where A_r is the accounting size for an HTTP redirect. Thus, a web server can improve its R and D by using DotSlash as follows. If it only uses HTTP redirect to offload client requests, its R is bounded by λ_d^m/A_r, and its D is bounded by $R(F+H)$. However, a web server can use DNS round robin to overcome this scaling limitation so as to further improve its R and D.

We performed experiments in our local area networks (LANs) and on PlanetLab [21] for two goals. First, given a web server with a constraint on its outbound bandwidth, we want to improve its R and D by using DotSlash rescue services, and aim to achieve an improvement close to the analytical bound, i.e., the web server can handle a request rate close to λ_d^m/A_r when only HTTP redirect is used. Second, we want to confirm that our workload control algorithm works as expected.

6.1 Workload Generation

We use *httperf* [18] to generate workloads. If the request rate to be generated is high, multiple httperf clients are used, each running on a separate machine. To simulate web hotspots, a small number of files are requested repeatedly from a web server. Each request uses a separate TCP connection. Thus, the request rate equals the connection rate. We made two enhancements to httperf to facilitate experiments on DotSlash. First, we extended httperf to handle HTTP redirects automatically since an httperf client needs to follow HTTP redirects in order to complete workload migrations from an origin server to its rescue servers. Second, we wrote a shell script to support workload profiles. A workload profile specifies a sequence of request rates and their testing durations, which is convenient for describing workload changes.

For a web server, its R and D are determined as follows. We use httperf clients to issue requests to the web server, starting at a low request rate, and increasing the request rate gradually until the web server gets overloaded. A client uses 7 seconds [9] as the timeout value for getting each response. If more than 10% [9] of issued requests time out, a client declares the web server as being overloaded. For a sequence of testing request rates that are monotonically increasing, $r_1 < r_2 < \cdots$, if the web server gets overloaded at r_i, then $R = r_{i-1}$. For all testing request rates, up to R, the maximum data rate delivered to clients is D.

6.2 Experimental Setup

In our LANs, we use a cluster of 30 Linux machines, which are connected using 100 Mb/s fast Ethernet. These machines have two different configurations, *CLIC* and *iDot*.

The former has a 1 GHz Intel Pentium III CPU, and 512 MB of memory, whereas the latter has a 2 GHz AMD Athlon XP CPU, and 1 GB of memory. They all run Redhat 9.0, with Linux kernel 2.4.20-20.9. PlanetLab consists of more than 300 nodes distributed all over the world, each with a CPU of at least 1 GHz clock rate, and has at least 1 GB of memory. PlanetLab nodes have four types of network connections: DSL lines, Internet2, North America commodity Internet, and outside North America. They all run Redhat 9.0, with Linux kernel 2.4.22-r3_planetlab, and PlanetLab software 2.0.

We set up the DotSlash software in three steps. First, we compile Apache 2.0.48 with the *worker* multi-processing module, the proxy modules, the cache modules, and our DotSlash module. We configure Apache as follows. Since reverse proxying is taken care of by DotSlash automatically, no proxy configuration is needed. Caching is configured with 256 KB of memory cache, and 10 MB of disk cache, and the maximum file size allowed in memory cache is 20 kB. For the DotSlash module, we only configure λ_d^m. Second, we use BIND 9.2.2 as the DNS server software, and set up a DNS domain *dot-slash.net*. All rescue servers register their virtual host names in this domain via dynamic DNS updates. Currently, we have tested DotSlash workload migrations via HTTP redirect, without using DNS round robin. Third, we set up a DotSlash service registry using an mSLP DA. Each web server registers itself with this well-known service registry, and discovers other web servers by looking up this registry.

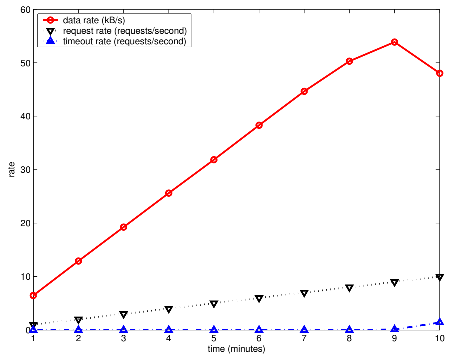
6.3 Experimental Results on PlanetLab

We run a web server on a PlanetLab DSL node, *planetlab1.gti-dsl.nodes.planet-lab.org* (referred to as *gtidsl1*), for which the outbound bandwidth is the bottleneck. We run httperf on a local CLIC machine. Ten files are requested repeatedly from *gtidsl1*, with an average size of 6 KB [30]. Our goal is to measure, from the client side, *gtidsl1*'s R and D in two cases, namely without using DotSlash versus using DotSlash.

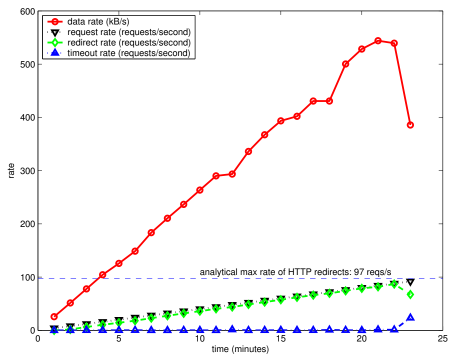
For the first case, DotSlash is disabled. The request rate starts at 1 request/second, increases to 20 requests/second, with a step size of 1, and each request rate lasts for 60 seconds. Fig. 7(a) shows the experimental results. In this figure, *gtidsl1* gets overloaded at 10 requests/second, where 14% of requests, 84 out of 600, time out. Thus, R is 9 requests/second. The measured D is 53.9 kB/s, attained when the request rate is R.

For the second case, DotSlash is enabled. We set *gtidsl1*'s λ_d^m to 53.9 kB/s. To provide needed rescue capacity for *gtidsl1*, we run another web server on a local *iDot* machine (named *maglev*), and its λ_d^m is set to 2000 kB/s. The request rate starts at 4 requests/second, increases to 200 requests/second, with a step of 4, and each request rate lasts for 60 seconds. Fig. 7(b) shows the experimental results. In this figure, when the request rate reaches 8 requests/second, the origin server *gtidsl1* starts to redirect client requests via HTTP redirects to the rescue server *maglev*. As the request rate increases, the redirect rate increases accordingly. Eventually, *gtidsl1* redirects almost all clients requests to *maglev*. In this experiment, *gtidsl1* gets overloaded at 92 requests/second, where 25% of requests, 1404 out of 5520, time out. Thus, R is 88 requests/second. The measured D is 544.1 kB/s, attained when the request rate is 84 requests/second.

Comparing the results obtained from the above two cases, we have $88/9 = 9.78$, and $544.1/53.9 = 10.1$, meaning that by using DotSlash rescue services, we got about an order of magnitude improvement for *gtidsl1* on its R and D, even if only HTTP



(a) Without using DotSlash rescue services



(b) Using DotSlash rescue services

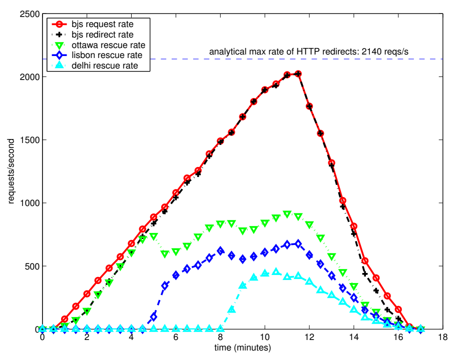
Fig. 7. The data rate and request rate for a PlanetLab DSL node *gtidsl1* in two cases, note different scales of ordinates

redirect is used. To show the effectiveness of DotSlash, we also compare R with its analytical bound λ_d^m/A_r below. In this experiment, we only measured λ_d^m at *gtidsl1*, without knowing its outbound bandwidth B. To be conservative, we use $U = (F + H)/(F+H+O) = 95\%$, where $F = 6$ KB, $H = 250$ bytes, and $O = 358$ bytes. Here the header overhead O for a single-request HTTP transaction is the same as that for an HTTP redirect (calculated in Section 4.3). Since the size of an HTTP redirect response is $n = 227$ bytes in the experiment, we have $A_r = (n + O)U = 556$ bytes. As a result, R is bounded by $\lambda_d^m/A_r = 53.9 * 1000/556 = 97$ requests/second, and we achieved $88/97 = 91\%$ of its analytical bound.

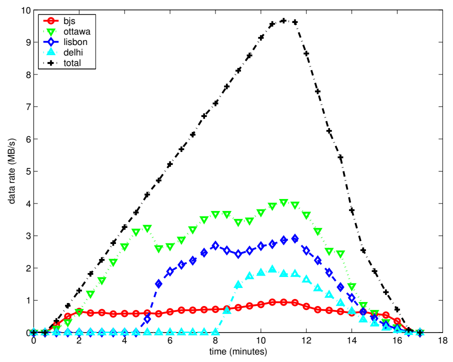
6.4 Experimental Results in LANs

In the previous section we have shown the performance improvement, measured from the client side, for a web server by using DotSlash rescue services in a wide area network setting. In this section we will show, via an inside look from the server side, how workload is migrated from an origin server to its rescue servers. The workload monitoring component in DotSlash maintains a number of counters for outbound HTTP traffic, including total bytes served, the number of client requests served, the number of client requests redirected, and the number of requests served for rescuing others. The values of these counters for a web server *host.domain* can be obtained from *http://host.domain/dotslash-status?auto*. By sampling these counters at a desired interval, we can calculate the needed average values of outbound data rate, request rate, redirect rate, and rescue rate.

In this experiment, four machines, *bjs*, *ottawa*, *lisbon*, and *delhi*, run as web servers, where *bjs* is an *iDot* machine, and the other three are *CLIC* machines. To emulate a scenario where *bjs* works as an origin server with a bottleneck on its outbound bandwidth, and the rest web servers work as rescue servers, we configured their λ_d^m as 1000, 7000, 5000, and 3000 kB/s, respectively. We run httperf on five *CLIC* machines, which



(a) The request rate and redirect rate at *bjs*, and the rescue rates at its rescue servers



(b) The data rate at each web server, and the total data rate of all web servers

Fig. 8. The request rates and data rates at the origin server *bjs* and its rescue servers

issue requests to *bjs* using the same workload profile. The maximum request rate is $400 * 5 = 2000$ requests/second, and the duration of the experiment is 15 minutes. Ten files are requested repeatedly, with an average size of 4 KB. We run a shell script to get the DotSlash status from the four web servers at an interval of 30 seconds. The retrieved status data are stored in round-robin databases using RRDtool [22], with one database for each web server. Fig. 8 shows the data rates and request rates for the four web servers in a duration of 17 minutes.

We observe the following results from Fig. 8(a). First, *bjs* can support a request rate of 2000 requests/second, which is close to λ_d^m/A_r, the analytical maximum rate of HTTP redirects at *bjs*. Since $A_r = (n + O)U = 468$ bytes in this experiment, where $n = 227$ bytes, $O = 358$ bytes, and U takes its default value 80%, we have $\lambda_d^m/A_r = 2140$ requests/second. Second, the redirect rate at *bjs* increases as the request rate increases, and it is roughly the same as the request rate once it is above 1500 requests/second. The reason is that *bjs* increases redirect probability P_r as its load increases. When the rate of HTTP redirects is greater than $\lambda_d^m \rho_n^u / A_r = 1603$ requests/second, P_r will stay at 1, that is all client requests are redirected from *bjs* to its rescue servers. Third, *bjs* allocates one rescue server at a time, and uses the one with the largest rescue capacity first. When a new rescue server is added in, the rescue rates at the existing rescue servers decrease. Also, the rescue rates at rescue servers are proportional to their rescue capacities because of the WRR at *bjs*.

Comparing Fig. 8(b) and 8(a), we observe that rescue servers have similarly shaped curves for their data rates and rescue rates. In contrast, as we expected, the origin server *bjs* have quite different shapes for its the request rate and data rate curves: its the request rate increases significantly from 200 requests/second at 1.5 minutes to 2000 requests/second at 11 minutes, but its data rate is roughly unchanged, staying at $\lambda_d^m \rho_n^u = 750$ kB/s for the most part. This indicates that *bjs* has successfully migrated its workload to its rescue servers under the constraint of its outbound bandwidth. Also, we observe that when the request rate is between 1600 and 2000 requests/second, the data rate at *bjs* goes above 750 kB/s, but still stays below $\lambda_d^m = 1000$ kB/s. This is because *bjs* can only support a rate of 1600 requests/second for HTTP redirects with a data rate of 750 kB/s. Furthermore, we observe that the total data rate of all web servers has a maximum value of 9.7 MB/s, which is higher than 9.2 MB/s, the maximum data rate measured from the httperf clients. The difference is due to our special accounting for HTTP redirects. As described in Section 4.3, an HTTP redirect is 227 bytes, but is counted as 468 bytes, which results in a rate increase of $241 * 2000 = 0.482$ MB/s for 2000 HTTP redirects.

7 Conclusion

We have described the design, implementation, and evaluation of DotSlash in this paper. As a rescue system, DotSlash complements the existing web server infrastructure to handle web hotspots effectively. It is self-configuring, scalable, cost-effective, easy to use, and transparent to clients. Through our preliminary experimental results, we have demonstrated the advantages of using DotSlash, where a web server increases the request rate it supported and the data rate it delivered to clients by an order of magnitude, even if only HTTP redirect is used.

We plan to perform trace-driven experiments on DotSlash by using log files from web hotspot events, and incorporate DNS round robin in the performance evaluation. Also, we plan to investigate load migration for dynamic content, which will extend the reach of DotSlash to more web sites. Our prototype implementation of DotSlash will be released as open source software.

References

1. T. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service (IWQoS)*, London, England, June 1999.
2. Stephen Adler. The slashdot effect: An analysis of three Internet publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.
3. Akamai homepage. http://www.akamai.com/.
4. Apache. HTTP server project. http://httpd.apache.org/.
5. M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable context-aware request distribution in cluster-based network servers. In *Annual Usenix Technical Conference*, San Diego, California, June 2000.
6. A. Barbir, Brad Cain, Raj Nair, and Oliver Spatscheck. Known content network (CN) request-routing mechanisms. RFC 3568, Internet Engineering Task Force, July 2003.
7. BitTorrent homepage. http://bitconjurer.org/BitTorrent/.
8. V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2):263–311, June 2002.
9. V. Cardellini, M. Colajanni, and P.S. Yu. Geographic load balancing for scalable distributed web systems. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, San Francisco, California, August 2000.
10. E. Coffman, P. Jelenkovic, J. Nieh, and D. Rubenstein. The Columbia hotspot rescue service: A research plan. Technical Report EE2002-05-131, Department of Electrical Engineering, Columbia University, May 2002.

11. Mark Day, Brad Cain, G. Tomlinson, and P. Rzewski. A model for content internetworking (CDI). RFC 3466, Internet Engineering Task Force, February 2003.
12. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.
13. E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.
14. J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *International World Wide Web Conference (WWW)*, Honolulu, Hawaii, May 2002.
15. K. Kong and D. Ghosal. Mitigating server-side congestion in the Internet through pseudoserving. *IEEE/ACM Transactions on Networking*, 7(4):530–544, August 1999.
16. W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at USENIX LISA, December 2001.
17. Q. Li and B. Moon. Distributed cooperative Apache web server. In *International World Wide Web Conference*, Hong Kong, May 2001.
18. D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. In *Workshop on Internet Server Performance (WISP)*, Madison, Wisconsin, June 1998.
19. Netcraft. Web server survey. http://news.netcraft.com/archives/web_server_survey.html.
20. V. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.
21. PlanetLab. http://www.planet-lab.org/.
22. RRDtool homepage. http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/.
23. P. Rzewski, Mark Day, and D. Gilletti. Content internetworking (CDI) scenarios. RFC 3570, Internet Engineering Task Force, July 2003.
24. Stan Schwarz. Web servers, earthquakes, and the slashdot effect. http://pasadena.wr.usgs.gov/office/stans/slashdot.html.
25. T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.
26. A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *IEEE International Conference on Network Protocols (ICNP)*, Paris, France, November 2002.
27. A. Vakali and G. Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, December 2003.
28. Paul Vixie, Sue Thomson, Y. Rekhter, and Jim Bound. Dynamic updates in the domain name system (DNS UPDATE). RFC 2136, Internet Engineering Task Force, April 1997.
29. Jia Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Review (CCR)*, 29(5), October 1999.
30. L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, Massachusetts, December 2002.
31. M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *USENIX Conference on Internet Technologies and Systems (USITS)*, Seattle, Washington, March 2003.
32. W. Zhao and H. Schulzrinne. Selective anti-entropy. In *ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.
33. W. Zhao, H. Schulzrinne, and E. Guttman. Mesh-enhanced service location protocol (mSLP). RFC 3528, Internet Engineering Task Force, April 2003.
34. W. Zhao, H. Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. Select and sort extensions for the service location protocol (SLP). RFC 3421, Internet Engineering Task Force, November 2002.