DotSlash: A Scalable and Efficient Rescue System for Handling Web Hotspots

Weibin Zhao and Henning Schulzrinne
Department of Computer Science
Columbia University
New York, NY 10027

{zwb,hgs}@cs.columbia.edu

ABSTRACT

This paper describes DotSlash, a scalable and efficient rescue system for handling web hotspots. 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 existing web server infrastructure such as CDNs to handle short-term load spikes effectively, but is not intended to support a request load constantly higher than a web site's planned capacity. DotSlash is scalable, cost-effective, easy to use, self-configuring, and transparent to clients. It targets small web sites, although large web site can also benefit from it. We have implemented a prototype of DotSlash on top of Apache. Experiments show that DotSlash can provide an order of magnitude improvement for a web server in terms of the request rate supported and the data rate delivered to clients even if only HTTP redirect is used. Parts of this work may be applicable to other services such as the Grid computational services and media streaming.

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 can be very bursty. Consider a well-identified problem – web hotspots, where a web site experiences a sudden and dramatic surge of request load. Web hotspots, also known as the Slashdot effect (i.e., mentioning a low-volume site in a high-volume site [2]) or the flash crowd phenomenon, pose a new challenge for designing scalable and efficient distributed web server systems. For web hotspots, a fixed set of dedicated servers suffers from two major problems. The first issue is efficiency. Web hotspots are infrequent events or even one-time event for most sites. This "15 minutes of fame" lasts for a short time (often in tens of minutes or a few hours), but may trigger a large load increase (the ratio of peak to average load may go up to 1000 [29]). Thus, it is not economical to overprovision a web site according to its peak load. Another issue with a fixed server set is scalability. Since the peak load is hard to predict, even overprovisioning is difficult. In other words, a planned capacity may turn out to be insufficient for an unexpected load. For example, although CNN.com has a well-planned capacity for its web site, it experienced a request load 20 times greater than its expected peak on September 11, 2001, which caused its web site to be overloaded for about three hours [19].

To handle web hotspots effectively, we advocate dynamic allocation of server capacity from a global server pool – those servers in the pool need to be distributed globally because the access link of a local network can become a bottleneck. As an example of global server pools, content delivery networks (CDN) [35] have been used by large web sites. However, small web sites often cannot afford the cost of CDN services particularly since they may need these services very rarely. Thus, 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* – a rescue system that supports dynamic collaborations among different web sites. Unlike server collaborations within a single web site, inter-web-site collaborations are more challenging in that we cannot assume a common administrator and cannot rely on static configuration to form a collaborating server group. As a rescue system, DotSlash continuously monitors the workload at a web server; when the server becomes heavily loaded, rescue services are activated to help the server to survive the load spike; and once the server's load returns to normal, the rescue services are phased out. As a result, a web site has a dynamic server set which includes a single or a cluster of fixed *origin servers* and a changing set of *rescue servers*. Rescue servers are drafted (and released later) from other sites dynamically by a site in reaction to its load changes. In this way, a web site can build an adaptive distributed web server system on the fly, and expand its capacity dynamically by utilizing spare capacity at other sites.

DotSlash is not designed to support a request load that is constantly higher than a web site's planned capacity, but rather serves as a complementary mechanism to 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 (e.g., capacity planning) should handle the request load sufficiently in most (e.g., 99.9%) cases, but they might

The work described in this paper was supported in part by the National Science Foundation under Grant No. ANI-0117738. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

be too expensive for short-term enormous load spikes (e.g., a 1000-fold increase), and might be insufficient for unexpected load increases. For these cases, rescue services such as DotSlash can intervene when needed so that a web site can support its request load in more (e.g., 99.999%) cases. In parallel, a web site can use service degradation [1] under heavily-loaded conditions. For example, turning off dynamic content and serving a trimmed version of static content can reduce CPU workload and network bandwidth consumption, which can help a web site to survive load spikes. As the last resort, a web site can use admission control [40] to prevent itself from being overloaded by rejecting a fraction of client requests and only admitting preferred clients.

DotSlash has the following advantages. First, it is scalable. A web site can expand its capacity as needed by using more rescue servers. Second, 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 existing web server infrastructure, without incurring any additional hardware cost. Third, it is easy to use. Standard DNS mechanisms such as DNS aliasing and round robin and HTTP redirect are used to offload client requests from an origin server to its rescue servers, without the need to change operating system or DNS server software. An add-on module to the web server software is sufficient to support all needed functions. Fourth, it is transparent to clients. As DotSlash only uses server-side mechanisms, client browsers remain unchanged, and client bookmarks continue to work. Fifth, it is self-configuring. Service discovery [15] is used to allow a web site to collaborate with other sites automatically, without any administrator intervention. 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.

Currently, DotSlash only supports load migration for static web pages. This is because in DotSlash a rescue server serves as a reverse caching proxy for its origin servers, and caching control for static web pages is relatively easy. Another reason is that a web site can turn off dynamic content by serving static pages under heavily-loaded conditions. We plan to investigate load migration for dynamic content in the next stage of this project.

As a rescue system for handling web hotspots, DotSlash targets small web sites. Large web sites can also benefit from DotSlash, but they often can afford more expensive mechanisms, such as dedicated server clusters or commercial CDN services. Parts of the work (such as self-configuration, dynamic collaboration, and load control via monitoring, prediction, offloading and feedback) may be applicable to other services, including computational services (Grid [13]), SIP proxy services [26] and media streaming, but synchronizing states (programs, registration entries and media files) may make this more difficult for short-term load spikes.

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 [38] 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 [35] 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 (e.g., in months), but less efficient for handling short-term (e.g., in hours) load spikes; and the cost of using CDN services is often too expensive for small web sites. Also, using CDN services needs advance configuration, 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 homogeneous (i.e., replicated) web servers deployed in a local area network (e.g., ScalaServer [5]) or strategically distributed in wide area networks (e.g., GeoWeb [8]), with a focus on load balancing among replicated servers 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, where the main goal is to relieve the heavily-loaded origin server by replicating its content dynamically to rescue servers and serving a fraction of client requests at rescue servers. DC-Apache [20] supports collaborations among heterogeneous web servers. However, it generates all hyperlinks dynamically by URI rewriting based on a document graph, which incurs a cost for each requested document, and it is less efficient to rebuild the document graph if documents are changing frequently. Also, DC-Apache relies on static configuration to form collaborating server groups, which limits its scalability and adaptivity to changing environments. DotSlash addresses these limitations by forming collaborating server groups dynamically, and using simpler and widely applicable mechanisms to offload client requests from origin servers. Along with the emerging of peer-to-peer (P2P) technology [25, 32, 27, 41], Backslash [30] suggests using P2P overlay networks to build distributed web server systems and using distributed hash table to locate resources.

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 [18] and CoopNet [23]. Clients can also form P2P overlay networks and use search mechanisms to locate resources. For example, PROOFS [31] employs randomization to build client-side P2P overlay networks, and BitTorrent [7] breaks large files into small parts for efficient retrieval. 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" [13], with a focus on large-scale computational problems and complex applications that involve many participants and different types of activities and interactions. 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 a emphasis on overload control at web servers and disseminating popular files to a large number of clients.

3. DOTSLASH OVERVIEW

3.1 Rescue Model

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 a load spike that exceeds its own capacity, a participating server discovers and uses spare capacities at other servers in its community via the DotSlash rescue services. In our current prototype, DotSlash is intended for a cooperative environment, and thus no payment is involved in obtaining rescue services. However, if authentication and payment mechanisms are incorporated, DotSlash could support commercial rescue services.

In DotSlash, a web server may benefit from rescue services or provide rescue services to others; but it does not involve itself with rescue services during normal times. Thus, 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. Note that these three states are mutually exclusive: a server is not allowed to get a rescue service from another server as well as to provide a rescue service to yet another server at the same time. In other words, a server cannot provide rescue services to others if itself is in the SOS state, and a server needs to shutdown all existing rescue services it provides before it requests rescue services from others. By enforcing this simple rule for provisioning and utilizing rescue services, we 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 a rescue relationship is set up between two servers, the one that benefits from the rescue service is the origin server, and the one that provides the rescue service is the rescue server. In other words, when a server is in the SOS state, it is the origin server for all of its rescue servers, and when a server is the rescue state, it is a rescue server for all of its origin servers. Note, however, a web server is always an origin server for its own, configured web sites. When it agrees to serve the content for a new, non-configured web site temporarily, it becomes a rescue server for that site.

Figure 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, i.e., S_x is the origin server, and S_y is the rescue server. 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. A web server, such as S_3, can simultaneously be a rescue server for multiple origin servers.

3.2 Rescue Examples

In DotSlash, a rescue server serves as a reverse caching proxy for its origin servers, and an origin server uses HTTP redirect and DNS round robin to offload client requests to its rescue servers. Thus, there are four rescue cases: (1) HTTP redirect and cache miss (i.e., the origin server uses HTTP redirect and the rescue server has a cache miss), (2) HTTP redirect and cache hit, (3) DNS round robin and cache miss, and (4) DNS round robin and cache hit. In case of a cache miss, the rescue server makes a reverse proxy request to the origin server. For simplicity, we only show examples for case 1 and 4; case 2 and 3 can be derived similarly.

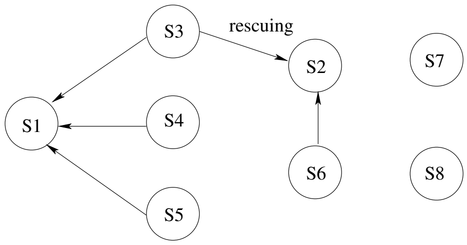


Figure 1: An example for DotSlash rescue relationships

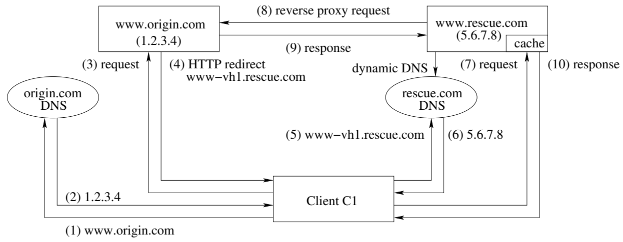


Figure 2: A rescue example for HTTP redirect and cache miss

Figure 2 shows an example for HTTP redirect and cache miss, where the origin server is *www.origin.com* with IP address *1.2.3.4*, the rescue server is *www.rescue.com* with IP address *5.6.7.8*, and the rescue server has assigned an alias *www-vh1.rescue.com* to the origin server. In this figure, a client C_1 follows a ten-step procedure to retrieve *http://www.origin.com/index.html*:

1. C_1 resolves the origin server's domain name *www.origin.com*;
2. C_1 gets the origin server's IP address *1.2.3.4*;
3. C_1 makes an HTTP request to the origin server using the URI *http://www.origin.com/index.html*;
4. C_1 gets an HTTP redirect from the origin server with the redirect URI *http://www-vh1.rescue.com/index.html*;
5. C_1 resolves the rescue server's alias *www-vh1.rescue.com*;
6. C_1 gets the rescue server's IP address *5.6.7.8*;
7. C_1 makes an HTTP request to the rescue server using the URI *http://www-vh1.rescue.com/index.html*;
8. As the request URI leads to a cache miss, the rescue server makes a reverse proxy request to the origin server using the URI *http://www.origin.com/index.html*;
9. Since an origin server always directly answers (i.e., never HTTP redirects) proxy requests from its rescue servers, the origin server sends the requested file to the rescue server;
10. The rescue server caches the requested file, and returns the file to C_1.

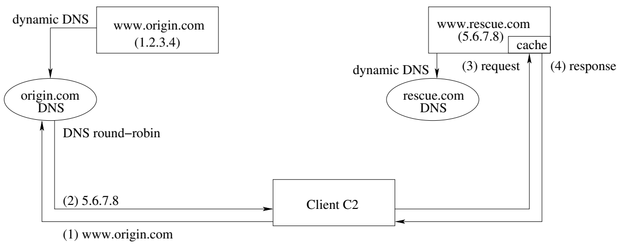


Figure 3: A rescue example for DNS round robin and cache hit

Figure 3 shows an example for DNS round robin and cache hit, where the setting is the same as that of the previous example, and the origin server has added the rescue server's IP address to its round robin local DNS. In this figure, a client C_2 follows a four-step procedure to retrieve *http://www.origin.com/index.html*:

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

4. DOTSLASH DESIGN

DotSlash has the following functional components: dynamic virtual hosting which allows a rescue server to serve the content of its origin servers on the fly, request redirection which allows an origin server to offload client requests to its rescue servers, workload monitoring which allows a web server to react quickly to load spikes, rescue control which allows a web server to properly interact with other web servers for handling web hotspots, and service discovery which allows servers of different web sites to learn about each other dynamically and collaborate automatically without any administrator intervention. Before discussing each component in turn below, we summarize the DotSlash parameters in Table 1.

4.1 Dynamic Virtual Hosting

Dynamic virtual hosting allows a rescue server to serve the content of its origin servers on the fly. As HTTP 1.1 [12] becomes widely deployed, virtual hosting (by using the *Host* header) has been supported by most web servers. However, existing virtual hosting such as Apache [4] needs advance configuration: registering virtual host names in DNS, creating DocumentRoot directories and files under those directories, and adding directives to the configuration file to map virtual host names to DocumentRoot directories. To support virtual hosting on the fly, DotSlash handles these configurations dynamically as follows.

A rescue server generates needed virtual host names dynamically by adding a sequence number component to its configured name. In Apache, the configured name of a web server is specified via the `ServerName` directive. If a rescue server has a configured name as *host.domain*, then its virtual host names are composed as *host-vh<seqnum>.domain*, where *<seqnum>* is monotonically increasing.

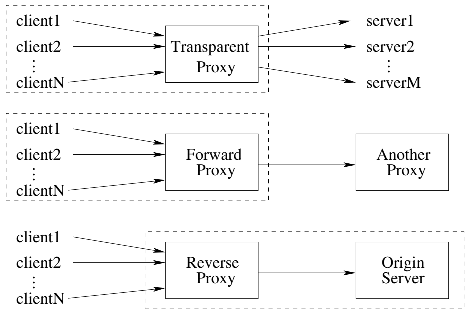


Figure 4: Three different types of proxy: transparent proxy, forward proxy, and reverse proxy

A rescue server registers its virtual host names using *A* records via dynamic DNS updates [36]. In addition to its local domain, a rescue server may register virtual host names in any domain as long as it uses a unique host name in the registration domain. Having a separate domain for virtual host name registrations is advantageous in that we don't need all rescue servers' local DNS to support dynamic DNS updates, which simplifies configuration requirements and reduces DNS related security risks. We have set up a domain *dotslash.cs.columbia.edu* (another domain *dot-slash.net* is ongoing), which accepts virtual host name registrations from all of our test machines, many of which are PlanetLab [24] nodes deployed across the whole world. For example, server *www.rescue.com* can obtain a unique host name *foo* in *dotslash.cs.columbia.edu*, and register virtual host names as *foo-vh<seqnum>.dotslash.cs.columbia.edu*.

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. To avoid potential confusion in mapping virtual host names, a rescue server does not reuse a virtual host name for two different origin servers.

A rescue server maintains a table to map each assigned virtual host name to its corresponding origin server. As a rescue server, *www.rescue.com* may receive requests that use three different kinds of *Host* header fields: its configured server 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*. The first case requests its configured content, and the last two cases request the content of its origin servers. Moreover, the second case is due to HTTP redirects from its origin servers, and the third case is due to DNS round robin when its origin servers have added its IP address to their local DNS. A rescue server adds a new entry to its mapping table whenever it provides a rescue service to a new origin server. 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 corresponding rescue service has been terminated, but rather keeps the mapping entry for a configured time such as a few hours or a few days, and redirects such a request back to the corresponding origin server via an HTTP redirect.

A rescue server supports dynamic content replications by working as a reverse caching proxy for its origin servers. For exam-

Parameter	Description	Unit	Default	Range
T_{low}	Low threshold below which release actions may be triggered		10%	$(0\%, 20\%]$
T_{alert}	Alert threshold above which initial allocation may be triggered		50%	$[40\%, 60\%]$
T_{high}	High threshold above which initial HTTP redirect is activated		75%	$[60\%, 80\%]$
T_{coff}	Cutoff threshold above which additional allocation is triggered		90%	$[80\%, 100\%)$
T_{redi}	Redirect threshold above which HTTP redirect is activated, adjusted dynamically			$[0\%, T_{high}]$
I_{ctrl}	Control interval for performing rate measurement and load control	seconds	1	
A_{redi}	Accounting size for an HTTP redirect response	bytes	468	
D_{out}	Maximum data rate for outbound HTTP traffic	kB/s		
d_{out}	Real data rate of all HTTP responses	kB/s		
d_{self}	Real data rate of HTTP responses for configured sites	kB/s		
D_{resc}	Maximum data rate of HTTP responses for rescuing other sites	kB/s		
d_{free}	Available data rate advertised for rescuing other sites	kB/s		
d_{resc}	Real data rate of HTTP responses for rescuing other sites	kB/s		
$d_{resc}(S_o)$	Real data rate of HTTP responses for rescuing S_o	kB/s		
d_{redi}	Real data rate of HTTP redirect responses	kB/s		
$D_{redi}(S_o, S_r)$	Maximum data rate redirected from S_o to S_r, adjusted dynamically	kB/s		
$d_{redi}(S_r)$	Real data rate redirected to S_r	kB/s		
R_{redi}	Rate of HTTP redirects	reqs/s		
D_{max}	Maximum data rate of HTTP responses delivered to clients	kB/s		
R_{max}	Maximum request rate supported	reqs/s		
F	Average size of requested files	KB		
U	Percentage of bandwidth that is usable for HTTP traffic		80%	

Table 1: Major DotSlash parameters, where kB is 1000 bytes, KB is 1024 bytes, and reqs/s is requests per second

ple, when a rescue server *www.rescue.com* has a cache miss for a request URI *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*. Figure 4 shows three different types of proxies: transparent proxy which forwards client requests to the corresponding web servers, forward proxy which forwards client requests to another proxy, and reverse proxy which forwards client requests to one or a small number of origin web servers. Using reverse caching proxy offers a few advantages for handling web hotspots. First, as files are replicated on-demand from the origin server to the rescue server, the origin server incurs low cost since it does not need to maintain states for replicated files and can avoid unnecessary transfers for 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. We focus on memory caching since a small number of hot files account for most accesses [16] during web hotspots, and it is likely that these hot files could fit into memory cache. Disk caching is used only for large files such as streaming media files.

4.2 Request Redirection

Request redirection [9, 6, 39] 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 fine grained load balancing. An origin server can achieve primitive load balancing in DNS by using multiple *A* records for its name as follows: when it has allocated a new rescue server, it adds the rescue server's IP address to its local DNS; and once it has released an existing rescue server, it removes the rescue server's IP address from its local DNS. Client requests are distributed to rescue servers during the origin server's name resolution phase via DNS round robin, which is transparent to clients, and incurs no additional delays for clients. However, DNS round robin can only provide crude load balancing due to DNS caching at clients and intermediate DNS servers. Using HTTP redirect, an origin server can more precisely control the load redirected to each rescue server and achieve better load balancing, but clients incur an extra round trip time for getting the requested content. Compared to the requested content – the container file plus embedded objects, an HTTP redirect is much smaller, typically less than 300 bytes. Thus, HTTP redirects can reduce in orders of magnitude the amount of data transferred at the origin server, which is useful when the outbound network bandwidth is the bottleneck. However, HTTP redirects cannot help much in reducing the CPU load because the origin server still needs to handle the same number of TCP connections.

As to the redirect URI in the *Location* header, we investigated three options: virtual directory, IP address, and virtual host name. The first option uses a virtual directory such as */dotslash-vh*. As an example, *http://www.origin.com/index.html* can be redirected as *http://www.rescue.com/dotslash-vh/www.origin.com/index.html*. A problem with this option is that it does not work for embedded relative URIs. The second option uses the IP address of the rescue server, which can save the client's DNS lookup time for the rescue server's name. However, this option suffers another problem in that the rescue server is unable to tell whether a received request is for itself or for one of its origin servers. We chose the last option, which uses a unique virtual host name of the rescue server. This option 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 a customized policy for HTTP redirects. To achieve fine grained load balancing, we use weighted round robin (WRR) as the basic policy for HTTP redirects, where the weight is the maximum data rate of HTTP redirects assigned by each rescue server. To improve cache hits for large files, we incorporate locality in the redirection policy for files larger than a configured size, which is achieved via consistent hashing [17] constrained by load. Due to factors such as caching and embedded relative URIs, the redirected load seen by the origin server may be different from the real rescue load served by the rescue server. For simplicity, an origin server only controls the load of redirected files, not including embedded objects such as images, to avoid parsing the redirected files, and relies on a load feedback from the rescue server to adjust its redirected load. We will discuss this control mechanism in Section 4.4.4.

Redirection needs to be avoided in some cases, such as communications between two collaborating servers and requests for retrieving server status information. On one hand, a request sender, either a web client or a web server, needs to bypass DNS round robin by using the server's IP address directly or a special domain name which is uniquely mapped to the server's IP address 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, that is 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 spikes. As different resources such as network bandwidth, CPU and memory at a web server may potentially become the bottleneck during a web hotspot, a separate workload metric is used for each resource, such as outbound data rate for network, load average for CPU, and the number of concurrent connections for memory. If any of these metrics exceeds its corresponding threshold, rescue actions are triggered; only when all metrics return to normal, rescue actions are phased out.

According to a recent study [23], network bandwidth is the most constrained resource for most web sites during hotspots. If we measure load spikes by two metrics: the data rate requested at a web server and the connection rate made to the web server, the first type of load spikes happens more frequently and occurs earlier than the second type. Thus, we focus on monitoring network bandwidth usage in DotSlash. Furthermore, we only monitor outbound HTTP traffic within a web server based on the following considerations. First, we assume that there is no significant other traffic besides HTTP at a web server. It is simple and self-contained for a web server to monitor HTTP traffic by itself, without relying on an external module to monitor traffic on the link. Second, we assume that 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. Normally a web server's outbound data rate is greater than its inbound data rate, thus it should be sufficient to only monitor outbound HTTP traffic.

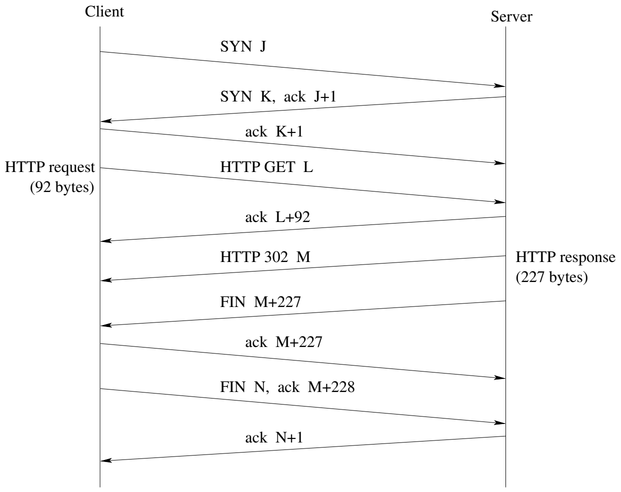


Figure 5: The client-server interaction for an HTTP redirect

Due to overhead such as TCP and IP headers, Ethernet header and trailer, and retransmissions, the HTTP traffic volume monitored by DotSlash is less than the real traffic volume on the link. As the header overhead is relatively constant and other overheads are usually small, to simplify calculation, we use U to indicate the percentage of bandwidth that is usable for HTTP traffic. Thus, if outbound bandwidth is B, then the maximum data rate for outbound HTTP traffic $D_{out} = B * U$. As we will show later in this section that the overhead for an HTTP transaction with a single request and response is below 400 bytes, $U = 80\%$ should be a good, although a bit conservative, estimate if the average size of HTTP responses is above 2 kB. This should be the case for most web sites when load is normal. However, as load increases, a web server will offload more client requests to its rescue servers via HTTP redirects. Since an HTTP redirect response is usually less than 300 bytes, a large number of HTTP redirects will make the average size of HTTP responses much smaller than 2 kB. Therefore, we need a special accounting for HTTP redirect responses.

To find out the overhead detail for an HTTP redirect, we use Ethereal [11] to capture needed packets at a web server, shown in Figure 5. For each HTTP redirect, the server sends five TCP packets: 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, the rest four TCP headers are 32 bytes each. The Ethernet header and trailer are 14 bytes and 4 bytes, respectively. The total overhead of TCP and IP headers plus Ethernet headers and trailers for five TCP packets is $(40+32*4)+20*5+(14+4)*5 = 358$ bytes. Thus, an HTTP redirect of size n bytes should be accounted as $A_{redi} = (n + 358) * U$ bytes. In our measurement, $n = 227$ bytes, $U = 80\%$, then $A_{redi} = (227 + 358) * 80\% = 468$ bytes.

4.4 Rescue Control

Rescue control allows a web server to properly interact with other web servers for handling web hotspots. We will discuss the following aspects of DotSlash rescue control: control strategies, rescue protocol, rescue server allocation, HTTP redirect control, and rescue relationship termination.

4.4.1 Control Strategies

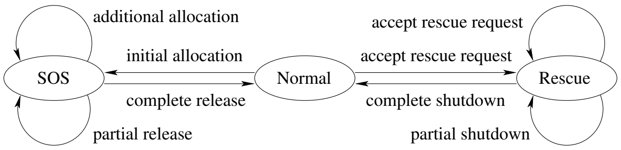


Figure 6: DotSlash state transitions

We associate rescue actions with load levels. A load level is the percentage of a measured load over the capacity (or a load limit), e.g., $100\%*d_{out}/D_{out}$ indicates the load level for outbound HTTP traffic in DotSlash. We distinguish different load levels using thresholds T_{low}, T_{alert}, T_{high}, and T_{coff}, which are defined in Table 1. Although these thresholds are configurable, for most web sites there should be no need to deviate from the default settings. Their associated actions will be discussed in Section 4.4.3, 4.4.4 and 4.4.5. When certain actions are triggered, a web server will change its state (defined in Section 3.1). Figure 6 summarizes the DotSlash state transitions.

We perform traffic rate measurement and load control in the granularity of the control interval I_{ctrl}, and represent other time intervals as TTL counters in terms of I_{ctrl}. A TTL counter is decreased by one every I_{ctrl} seconds. Once reaching zero, it is restored to its initial value, and the associated actions are triggered. For example, if service discovery needs to be performed every 10 minutes, and $I_{ctrl} = 5$ seconds, then the TTL for service discovery is 120.

For outbound HTTP traffic, we measure the outbound data rate d_{out}, the self data rate d_{self}, the rescue data rate d_{resc}, and the redirect data rate d_{redi}. Also, the rescue data rate is measured for each origin server, such as $d_{resc}(S_{o_1})$ and $d_{resc}(S_{o_2})$, and the redirect data rate is measured for each rescue server, such as $d_{redi}(S_{r_1})$ and $d_{redi}(S_{r_2})$. For any web server, $d_{out} = d_{self} + d_{resc}$, and $d_{redi} \leq d_{self}$. For any origin server, $d_{resc} = 0$.

We handle load increases and decreases differently. In order to react quickly to load spikes, rescue actions are triggered, such as allocating rescue servers and performing HTTP redirects, only based on the current load level. In contrast, to keep system stable, rescue services are phased out, such as releasing rescue servers, only when the load levels are consistently low. A load level is *consistently low* if it is below T_{low} for N_{low} (configurable and defaulting to 30) number of consecutive control intervals.

4.4.2 Rescue Protocol

Servers of different web sites collaborate with each other using the DotSlash rescue protocol. It is an application-level protocol that uses single-line (ending with '$\backslash n$') pure text messages. It is a request-response protocol since each request triggers a response. Each message starts with a message sequence number <*msg_seq_no*>, which is used to match a response with a request. A request has a command string <*cmd_str*> followed by optional parameters, where <*cmd_str*> starts with a letter: [a-zA-Z], and is case insensitive. A response has a response code <*rep_code*> followed by the corresponding response string <*rep_str*> and optional parameters, where <*rep_code*> is a three-digit code starting with [0-9]. Therefore, we can distinguish a <*cmd_str*> from a <*rep_code*>, and thus identify whether a message is a request or a response.

Unlike a traditional client-server protocol where a host is either a client sending requests, or a server sending responses, in DotSlash both origin servers and rescue servers can initiate requests. Currently, DotSlash only defines three command strings: SOS for initiating a rescue relationship, RATE for adjusting a redirect data rate, and SHUTDOWN for terminating a rescue relationship. An SOS command is always sent by an origin server, and a RATE command is always sent by a rescue server, but a SHUTDOWN command may be sent by an origin server or a rescue server. Figure 7 gives two examples for the DotSlash rescue protocol, where $S_o \rightarrow S_r$ denotes that the message is sent from the origin server S_o to the rescue server S_r. Example 1 shows the establishment of a rescue relationship between S_o and S_r, the adjustments of the HTTP redirect data rate, and termination of the rescue relationship. Example 2 shows that a rescue request from S_o is rejected by S_r.

\# Example 1
$S_o \rightarrow S_r$: 1 SOS www.origin.com 1.2.3.4 8080 600
$S_r \rightarrow S_o$: 1 200 OK www-vh1.rescue.com 5.6.7.8 8080 100
$S_r \rightarrow S_o$: 1 RATE 60
$S_o \rightarrow S_r$: 1 200 OK
$S_r \rightarrow S_o$: 2 RATE 50
$S_o \rightarrow S_r$: 2 200 OK
$S_r \rightarrow S_o$: 3 RATE 100
$S_o \rightarrow S_r$: 3 200 OK
$S_o \rightarrow S_r$: 2 SHUTDOWN
$S_r \rightarrow S_o$: 2 200 OK

\# Example 2
$S_o \rightarrow S_r$: 1 SOS www.origin.com 1.2.3.4 8080 600
$S_r \rightarrow S_o$: 1 403 Reject

Figure 7: Examples for DotSlash rescue protocol

4.4.3 Rescue Server Allocation

Due to negotiation delays, allocating a new rescue server is performed in advance before the corresponding request redirections are needed. We distinguish two types of rescue server allocations: initial allocation and additional allocation. An *initial allocation* is triggered if a web server is in the normal state, and its $d_{out}/D_{out} > T_{alert}$. An initial allocation needs to be completed before the server's d_{out}/D_{out} reaches T_{high}, which is the threshold when the server starts offloading client requests via HTTP redirects. An *additional allocation* is triggered if an origin server has rescue servers, and the HTTP redirect load levels of all these rescue servers have reached T_{coff}. An additional allocation needs to be completed before the HTTP redirect load levels of existing rescue servers approach 100% so that the origin server can always have a rescue server to which needed client requests can be offloaded via HTTP redirects.

To initiate a rescue relationship, an origin server sends an SOS command to a chosen rescue server candidate. The command has the following parameters: the origin server's fully qualified domain name, its IP address, its port number for web requests, and the maximum idle time (in seconds) after which the rescue server should trigger an idle shutdown (see Section 4.4.5). 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 maximum HTTP redirect data rate (in kB/s) that the origin server can offload to the rescue server.

In DotSlash, rescue servers are allocated based on the following considerations. First, an origin server should keep the number of its rescue servers as small as possible. Since a rescue server is a reverse caching proxy for the origin server, a cache miss at a rescue server will trigger a file transfer from the origin server. Minimizing the number of rescue servers can potentially reduce their cache misses, and thus reduce the data transfers at the origin server. Therefore, a rescue server with large rescue capacity is preferred over several smaller ones. Second, as allocating rescue servers involve delays, an origin server performs at most one round of allocations in each I_{ctrl}.

4.4.4 *HTTP Redirect Controls*

DotSlash performs two HTTP redirect controls. The first one is that an origin server activates HTTP redirects once $d_{out}/D_{out} > T_{redi}$, and dynamically adjusts T_{redi} as follows:

$$T_{redi} = T_{high} - R_{redi} * A_{redi}/D_{out}$$
$$\text{if } T_{redi} < 0 \text{ then } T_{redi} = 0$$

where R_{redi} is the rate of HTTP redirects in the previous control interval. If $R_{redi} = 0$, $T_{redi} = T_{high}$; if R_{redi} increases, T_{redi} decreases; and if $R_{redi} > D_{out} * T_{high}/A_{redi}$, $T_{redi} = 0$, where an origin server offloads all received client requests via HTTP redirects. The maximum rate of HTTP redirects at an origin server is bounded by D_{out}/A_{redi}.

The second HTTP redirect control is that the rescue server S_r dynamically adjusts $D_{redi}(S_o, S_r)$ for its origin server S_o. First, S_r allocates a data rate $D_{alloc}(S_o)$ to S_o, and sets $D_{init}(S_o)$ to $D_{alloc}(S_o) * T_{coff}$. Then, it sets $D_{redi}(S_o, S_r)$ to $D_{init}(S_o)$, and indicates this value in its "200 OK" response to S_o's SOS request. Afterwards, it adjusts $D_{redi}(S_o, S_r)$ as follows:

$$\begin{array}{ll} \text{if} & d_{resc}(S_o) > D_{alloc}(S_o) \\ \text{then} & D_{redi}(S_o, S_r) = D_{redi}(S_o, S_r)/(d_{resc}(S_o)/D_{alloc}(S_o)) \\ \text{else} & \text{if } \; d_{resc}(S_o) < D_{init}(S_o) \text{ and } D_{redi}(S_o, S_r) < D_{init}(S_o) \\ & \text{then } D_{redi}(S_o, S_r) = D_{init}(S_o) \end{array}$$

Whenever $D_{redi}(S_o, S_r)$ is changed, S_r sends a RATE request to S_o to adjust its allowed HTTP redirect data rate. On the other hand, S_o ensures that its $d_{redi}(S_r)$ is below $D_{redi}(S_o, S_r)$.

4.4.5 *Rescue Relationship Termination*

A rescue relationship is terminated by sending a SHUTDOWN request either from the origin server or from the rescue server. We call it a *rescue release* when a rescue relationship is terminated by the origin server. We distinguish two types of rescue releases: partial release and complete release. An origin server triggers a *partial release* if it has multiple rescue servers, and the HTTP redirect load levels of all these rescue servers are consistently low. In a partial release, only one rescue server is released, and the released rescue server has the smallest D_{redi}. An origin server triggers a *complete release* if its d_{out}/D_{out} is consistently low, at which point all remaining rescue servers are released.

We call it a *rescue shutdown* when a rescue relationship is terminated by the rescue server. We distinguish two types of rescue shutdowns: idle shutdown and overload shutdown. A rescue server triggers an *idle shutdown* if it has not served any rescue traffic for an origin server for more than MaxIdle seconds, where MaxIdle is specified in the origin server's SOS request, and defaults to $10 * N_{low} * I_{ctrl}$ seconds. The goal of an idle shutdown is to release rescue resources in case of the origin server failure or network separation. A rescue server triggers an *overload shutdown* if $d_{self}/D_{out} > T_{alert}$, or $d_{resc}/D_{out} > T_{high}$, or $d_{out}/D_{out} > T_{coff}$. We also distinguish partial shutdowns from complete shutdowns. While a rescue server terminates all its rescue relationships in a *complete shutdown*, it only terminates part of its rescue relationships in a *partial shutdown*.

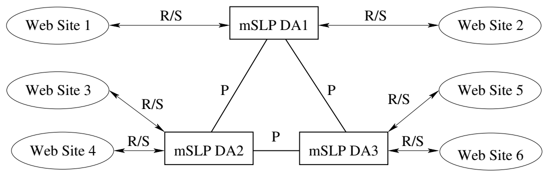


Figure 8: DotSlash service registry architecture, where R/S denotes register/search, and P denotes a peer relationship.

4.5 Service Discovery

Service discovery allows servers of different web sites to learn about each other dynamically and collaborate automatically without any administrator intervention. We will discuss the following aspects of service discovery in DotSlash: service discovery protocol, service registry architecture, service description, service advertisement, and service search.

Different service discovery protocols [15, 37, 34, 33] have been developed in recent years. We chose the Service Location Protocol (SLP) [15] as the DotSlash service discovery protocol since SLP is an IETF (Internet Engineering Task Force) proposed standard for service discovery in IP networks, and it is flexible, lightweight and powerful.

Figure 8 shows the DotSlash service registry architecture, which features multiple well-known service registries that maintain a fully-meshed peer relationship. This architecture is based on our previous work on the SLP mesh enhancement (mSLP [42]). Although SLP is designed for service discovery within one administrative domain, any domain can set up an SLP Directory Agent (DA) as a DotSlash service registry for the public good, which accepts service registrations and answers service search queries from web servers in any domain. 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 [44] 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 only serve the scope "DotSlash" (reserved for the DotSlash rescue services) so that these DAs will not affect local service discovery.

A service template is defined to describe the DotSlash rescue services, which has the following service attributes: URL, IP address, PortDots, and d_{free}. The URL gives the web server's domain name. If the web server does not use port 80, the URL must specify a port number. The IP address is used to bypass DNS round robin. PortDots specifies the port number for the DotSlash rescue services. d_{free} specifies the available data rate for rescue traffic. The maximum data rate for rescue traffic $D_{resc} = D_{out} * (1 - T_{alert})$, which defaults to $D_{out} * 50\%$ since T_{alert} defaults to 50%. A web server updates d_{free} dynamically as follows. If it is in the normal state, $d_{free} = D_{resc}$; if it is in the SOS state, $d_{free} = 0$; and if it is in the rescue state, d_{free} equals D_{resc} minus the allocated data rate for rescue traffic.

Service registrations are performed by using an SLP service agent. All registrations are soft states at DotSlash registries. A web server makes service registrations periodically. But once d_{free} is changed, the web server makes a new registration immediately.

Service searches are performed by using an SLP User Agent. To get ready for load spikes, a web server performs service searches periodically, and maintains a list of rescue server candidates. A DotSlash service search request uses preference filters [43] and the attribute list extension [14]. Preference filters allow the registry to sort the search result based on available rescue capacity and to only return the desired number of matching entries, which is useful if many entries match a search request. The attribute list extension allows the registry to return matching URLs and other service attributes in one response. An origin server should choose a rescue server that has an equivalent or larger rescue capacity than its own capacity since a rescue server with too small a capacity cannot help much. Also, an origin server should diversify its rescue servers (e.g., in different administrative domains) to avoid that they experience peak loads at the same time.

5. IMPLEMENTATION

5.1 Architecture

We chose Apache [4] as our base system since it is open source and is the most popular web server [22]. We compiled Apache with the *worker* multi-processing module (MPM), the proxy module, and the cache module. Figure 9 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, which is initialized when the Apache server is started. Mod_dots issues DotSlash commands to Dotsd via UDP to request Dotsd to allocate rescue servers or to terminate existing rescue services provided to others. DNS servers and DotSlash service registries are DotSlash components external to the Apache server. Dotsd interacts with DotSlash service registries using SLP, and interacts with DNS servers using the DNS protocol. We use BIND as DNS servers, which supports DNS round robin and dynamic DNS updates. We use mSLP DAs as DotSlash service registries. A web server interacts with other web servers via its Dotsd using the DotSlash rescue protocol (see Section 4.4.2) carried by TCP.

5.2 Control Data in Shared Memory

We use shared memory to store control data structures that are accessed by both Mod_dots and Dotsd. These data structures are divided into two parts: a traffic meter and a peer table. The traffic meter keeps traffic accounting information for the web server itself. The peer table maintains information for DotSlash peers, which is used in rescue server selection and virtual host name mapping. A web server's DotSlash peers are its collaborating web servers via the DotSlash rescue services. Based on our rescue model from Section 3.1, the DotSlash peers of an origin server are all rescue servers, and the DotSlash peers of a rescue server are all origin servers. The peer table also keeps traffic accounting information for DotSlash peers such that an origin server and a rescue server can control d_{redi} and d_{resc}, respectively, for each peer.

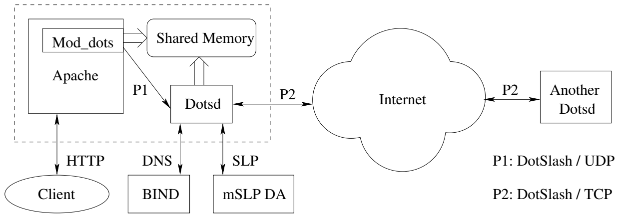


Figure 9: DotSlash software architecture

We perform traffic accounting in two time intervals: the current control interval, that is the most recent I_{ctrl} seconds, and the server's lifetime, that is from the server's booting time until now. The former accounting calculates various rates for current traffic, which are used to trigger needed rescue actions. The corresponding counters are reset to zero at the end of current control interval. The latter accounting allows calculating the average traffic rates by sampling the corresponding counters at the desired time intervals.

5.3 Dotsd

Dotsd is implemented using pthread. It has three main threads: a control thread which runs at the end of each control interval for processing tasks that need to be done periodically, a UDP server which processes requests from the local Mod_dots, and a TCP server which 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 (SA) for service registrations, and an SLP User Agent (UA) for service searches. We discuss them in turn below.

5.3.1 *The Control Thread*

The control thread runs every I_{ctrl} seconds to perform the following tasks. First, it checks current and recent traffic rates to determine whether it needs to terminate one or all rescue relationships: partial or complete release for an origin server, or idle shutdown for a rescue server (see Section 4.4.5). If a rescue relationship needs to be terminated, a SHUTDOWN command is issued to the corresponding DotSlash peer. Second, the control thread checks the rescue data rate d_{resc} for each origin server, if it needs to adjust D_{redi} for an origin server, a RATE command is issued to that origin server (see Section 4.4.4). Third, the control thread resets all control interval related counters to zero. Finally, it checks whether it needs to do service discovery. If the service registration timer is expired, it activates the SLP SA to perform a new service registration; if the service search timer is expired, it activates the SLP UA to perform a new service search.

5.3.2 *The UDP Server*

The UDP server processes the following commands issued by the local Mod_dots: an ALLOC command for initial and additional rescue server allocations (see Section 4.4.3), and a SHUTDOWN command for overload shutdown (see Section 4.4.5). To allocate a new rescue server, the UDP server chooses a web server from the rescue server candidate table, makes a TCP connection to the chosen server at PortDots, issues an SOS request, and creates a new thread, denoted as a TCP handler, for handling further interactions with the rescue server. To perform an overload shutdown, the UDP server issues a SHUTDOWN request to each origin server.

5.3.3 The TCP Server

The TCP server accepts connections from other Dotsds and creates a new thread acting as a TCP handler for processing each accepted connection. A TCP handler interacts with another TCP handler at a different Dotsd in a peer-to-peer way. Both can issue requests as well as process requests and responses from each other. As described in Section 4.4.2, a TCP handler may issue or process the following requests: SOS, RATE, and SHUTDOWN. When a new rescue relationship is established, the peer information is added to the peer table in shared memory (see Section 5.2). The origin server also adds the rescue server's IP address to it local DNS to allow offloading client requests via DNS round robin. When a rescue relationship is terminated, the origin server removes the rescue server from its peer table, and removes the rescue server's IP address from its local DNS. Instead of removing the origin server immediately from its peer table, the rescue server changes the origin server's peer type from *origin* to *expired origin*, and keeps the entry for a configurable period (see Section 4.1).

5.3.4 The DNS Client

The DNS client performs dynamic DNS updates. An origin server uses the DNS client to add or remove a rescue server's IP address from its local DNS so that offloading client requests via DNS round robin to the rescue server can be enabled or disabled. A rescue server uses the DNS client to add new aliases to its local DNS or a chosen domain. We do not need any new IP addresses for the above DNS updates.

5.3.5 The Service Registration Client

An SLP Service Agent is used for registering the web server to a DotSlash service registry. A new registration is triggered by the control thread if the registration timer has expired, or by the UDP server or a TCP handler if the available data rate for rescue traffic d_{free} has changed.

5.3.6 The Service Search Client

An SLP User Agent is used for searching available rescue servers at a DotSlash service registry. A new search is triggered by the control thread if the search timer has expired. The search results are used to build a list of rescue server candidates.

5.4 Mod_dots

Mod_dots constantly monitors the web server's workload, and triggers rescue actions based on current traffic rates. It performs HTTP redirects for an origin server, and supports dynamic virtual hosting for a rescue server. Mod_dots also implements a content handler to provide the current DotSlash status for the web server as a dynamically generated web page.

5.4.1 Workload Monitoring

We use an Apache output filter to compute the length of each HTTP response as the sum of the content length and the HTTP header length. The output filter is applied to each connection, and is registered in the Apache *pre_connection* phase. The length of the current HTTP response is calculated and recorded in a per-connection data structure for Mod_dots. In the Apache *log_transaction* phase, the length of the current HTTP response is added to related accounting counters, then it is reset to zero. For HTTP redirects, we perform a special accounting as described in Section 4.3 to take the different overhead into account.

Mod_dots triggers rescue actions based on current traffic rates. If a web server is in the normal state and its $d_{out}/D_{out} > T_{alert}$, then an initial allocation of rescue servers is triggered. For a rescue server, if its $d_{self}/D_{out} > T_{alert}$, or $d_{resc}/D_{out} > T_{high}$, or $d_{out}/D_{out} > T_{coff}$, then an overload shutdown is triggered.

5.4.2 HTTP Redirects

An origin server starts offloading client requests via HTTP redirects if $d_{out}/D_{out} > T_{redi}$. All HTTP redirects are performed in the Apache *post_read_request* phase.

To perform an HTTP redirect, Mod_dots picks a rescue server from its DotSlash peer table based on the following redirect policy. If the requested file is large, consistent hashing [17] is used to associate the request URI with an existing rescue server; otherwise, the least loaded rescue server is selected. Other policies can be incorporated later, such as taking into account the proximity of the client to rescue servers to reduce client-perceived delays. Next, Mod_dots updates traffic accounting information for the chosen rescue server by adding the requested file size to the corresponding counters. Then, Mod_dots checks whether it needs to trigger an additional allocation of rescue servers (see Section 4.4.3). At last, Mod_dots uses the chosen rescue server's virtual host name and its HTTP port number to fill in the host part of the redirect URI. The rest part of the redirect URI is copied from the origin request URI. For example, if a request URI is *http://www.origin.com/index.html*, the chosen rescue server is *www-vh1.rescue.com*, and its HTTP port is 8080, then the redirect URI is *http://www-vh1.rescue.com:8080/index.html*.

An origin server does not offload a client request via an HTTP redirect if the request's client IP address is the same as a rescue server's IP address (i.e., it is a reverse proxy request from a rescue server), or if the request is to get the web server's DotSlash status page.

5.4.3 Dynamic Virtual Hosting

Mod_dots supports dynamic virtual hosting for a rescue server by mapping virtual host names dynamically, converting a client request to a reverse proxy request to the origin server dynamically, and performing caching controls.

For a rescue server, Mod_dots maps the host name in the request URI dynamically in the Apache *post_read_request* phase. Because of DNS round robin and HTTP redirect, the host name in a request URI can be an origin server name or the virtual host name assigned to an origin server. Thus, Mod_dots compares the host name in the request URI with each origin server name and each assigned virtual host name in its DotSlash peer table (see Section 4.1). This comparison has three possible results: there is no match, then the request will be processed as normal; an origin server matches, then the request will trigger a reverse proxy to the origin server in case of a cache miss for the request URI; or an expired origin server matches, then the request will be redirected to the origin server via an HTTP redirect.

If there is a cache hit for a request URI, Mod_dots serves the response from its cache; otherwise, Mod_dots converts the request to a reverse proxy request to the origin server in the Apache *translate_name* phase, and stores the requested file in the cache. Note that Mod_dots takes care of reverse proxying dynamically; there is no need to use any proxy directives in the Apache configuration file *httpd.conf*.

DotSlash caching controls are built on top of existing Apache caching mechanisms. In DotSlash, a rescue server is a reverse caching proxy for its origin servers. To minimize cache misses and thus minimize the number of reverse proxy requests, all processes at a rescue server should share the same memory cache. In the current version of Apache 2.0.48, however, each process has its own memory cache. Instead of rewriting the existing Apache *mem_cache* module and implementing memory cache in shared memory, we use a simple solution as follows. First, we chose the Apache *worker* MPM, which is a hybrid multi-threaded multi-process MPM: each process has a fixed number of threads (say, 25); and normally only a small number of processes is needed (say, 2). As the number of memory caches is equal to the number of processes, the issue of memory cache sharing among processes becomes less important than that among threads within the same process. Second, we apply cache control to threads within the same process to avoid concurrent reverse proxy requests for the same URI, which improves cache efficiency at a rescue server, and reduces the workload at the corresponding origin server. Since it takes some time for a reverse proxy request to complete, and fill the requested file in the memory cache, Mod_dots maintains a table of ongoing reverse proxy URIs. In the Apache *post_read_request* phase, Mod_dots handles a URI that may trigger a reverse proxy request as follows:

```
if      the URI is not in the cache
then if      the URI is in the ongoing reverse proxy table
     then the thread waits until the ongoing reverse proxy
          request for the URI has completed
     else the thread puts the URI into the ongoing reverse
          proxy table and will initiate a reverse proxy
          request for the URI in a later phase
```

In the Apache *log_transaction* phase, if a thread has performed a reverse proxy request, it signals all threads that are waiting on the URI. A URI is removed from the ongoing reverse proxy table when it has completed and no thread is waiting on it.

5.4.4 DotSlash Status Display

Mod_dots 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

6.1 Workload Generation

We use *httperf* [21] to generate needed workloads. 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. If the request rate to be generated is high, multiple httperf clients are needed, each running on a different machine.

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.

6.2 Performance Metrics

We use two performance metrics to evaluate a web server, D_{max} the maximum data rate (in kB/s) of HTTP responses delivered to clients, and R_{max} the maximum request rate (in requests/second) supported. Our goal is to improve a web server's D_{max} and R_{max} by using the DotSlash rescue services.

For a web server without using DotSlash, its $D_{max} = D_{out}$, and $R_{max} = D_{out}/F$, where F is the average size of requested files (in KB). We ignore the header size of HTTP responses here, which is relatively small (about 250 bytes) compared to F. We assume that the CPU is not a bottleneck. By using DotSlash, the web server can improve its R_{max} and D_{max}. If only HTTP redirect is used to offload client requests, R_{max} is bounded by D_{out}/A_{redi}, and D_{max} is bounded by $R_{max} * F$. If DNS round robin is used as well, R_{max} and D_{max} can be improved further; the degree of improvement depends on factors such as client distribution and DNS caching.

For a web server, its R_{max} and D_{max} are determined as follows. We use one or multiple httperf clients to issue requests to the web server, starting at a low request rate, then increasing the request rate gradually, until the web server gets overloaded. A client uses 7 seconds [8] as the timeout value for getting the response from the web server for an issued request. If more than 10% [8] of issued requests time out, a client declares the web server as overloaded. We refer to the first request rate under which the web server gets overloaded as the overloading rate, and the testing request rate just before the overloading rate is R_{max}. For all testing request rates, up to R_{max}, the maximum data rate delivered to clients is D_{max}.

6.3 Experiments

6.3.1 Goals of Experiments

The goals of our experiments on DotSlash are as follows. First, given a web server with a constraint on its outbound bandwidth, we want to improve its R_{max} and D_{max} by using the DotSlash rescue services, and aim to achieve an improvement close to the analytical bound. Second, we want to confirm that our workload control algorithm works as expected, which can properly handle a request rate up to D_{out}/A_{redi} when only HTTP redirect is used.

6.3.2 Experimental Setup

We performed experiments in our local area networks (LAN) and on PlanetLab [24]. 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. On PlanetLab, we have more than 300 nodes over the whole world, each has 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 compiled Apache 2.0.48 with the worker MPM, the proxy modules, and the cache modules, and added the DotSlash module (Mod_dots) and DotSlash daemon (Dotsd) to it to support the DotSlash rescue services. As reverse proxying is taken care of by Mod_dots automatically, proxy configuration is not needed. The Apache 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. Regarding the DotSlash rescue services, we only configured D_{out} for each web server; other parameters take their default values. Second, we chose BIND 9.2.2 as the DNS server software, and set up a DNS domain *dotslash.cs.columbia.edu*. All rescue servers register their virtual host names in this domain via dynamic DNS updates. Currently, we only tested DotSlash workload migration via HTTP redirect, without using DNS round robin. Third, we set up a DotSlash service registry using an mSLP Directory Agent. If multiple DotSlash service registries are needed for reliability and scalability, they can be supported easily by using the SLP mesh enhancement [42]. Each web server registers itself with this well-known service registry, and discovers other web servers by looking up this registry. The rescue capacity a web server provided is D_{resc}, which defaults to $D_{out} * 50\%$ (see Section 4.5).

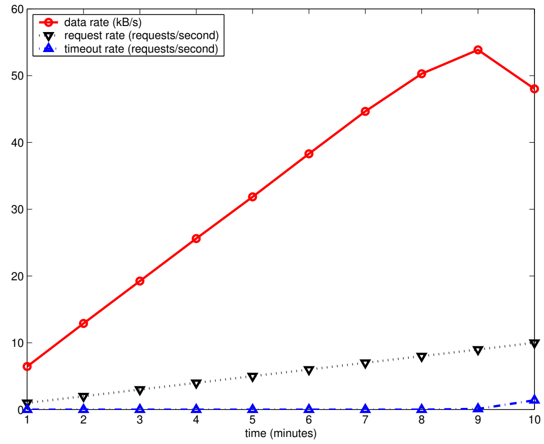
6.3.3 Experimental Results on PlanetLab

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 [39]. Our goal is to measure, from the client side, *gtidsl1*'s R_{max} and D_{max} in two cases, namely without using DotSlash versus using DotSlash.

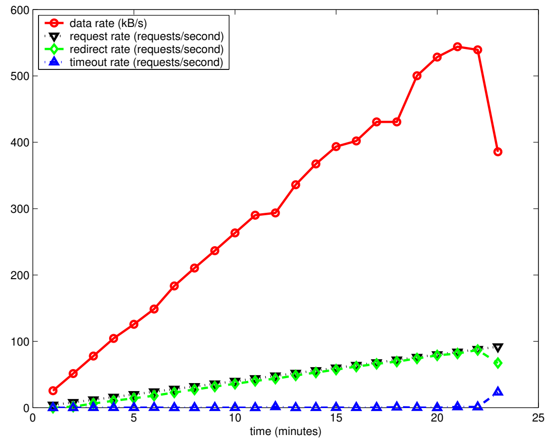
For the first case, DotSlash is disabled. The request rate starts at 1 request/second, increases to 20 requests/second, with a step of 1, and each request rate lasts for 60 seconds. Figure 10(a) shows the experimental results. In this figure, *gtidsl1* gets overloaded at 10 requests/second, where 14%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

For the second case, DotSlash is enabled. We set *gtidsl1*'s D_{out} to 53.9 kB/s. To provide needed rescue capacity for *gtidsl1*, we run another web server on a local iDot machine (referred to as *maglev*), and its D_{out} 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. Figure 10(b) shows the experimental results. In this figure, the origin server *gtidsl1* starts redirecting client requests, via HTTP redirects, to the rescue server *maglev* when the request rate reaches 8 requests/second. 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_{max} is 88 requests/second. The measured D_{max} is 544.1 kB/s, which appears when the request rate is 84 requests/second.

Comparing the results obtained from the above two cases, we have $88/9 = 9.78$, and $544.1/53.9 = 10.1$, which means that by using the DotSlash rescue services, we got about an order of magnitude improvement for *gtidsl1* on its R_{max} and D_{max}, even if only HTTP redirect is used. In this experiment, we only measured D_{out} at *gtidsl1*, without knowing its outbound bandwidth B. Using $U = 80\%$ is likely to over estimate B. Based on the calculation we did in Section 4.3, the overhead for an HTTP transaction is 358 bytes, and the bandwidth consumption of an HTTP redirect is 585 bytes. If F is 6 KB, and the average header size of HTTP responses is 250 bytes, then an HTTP response is 6394 bytes in average. To be conservative, we use $U = 6394/(6394 + 358) = 95\%$, and estimate A_{redi} as $585 * 95\% = 556$ bytes. As a result, the request rate is bounded by $D_{out}/A_{redi} = 53.9 * 1000/556 = 97$ requests/second, and we achieved $88/97 = 91\%$ of its analytical bound.



(a) Without using the DotSlash rescue services

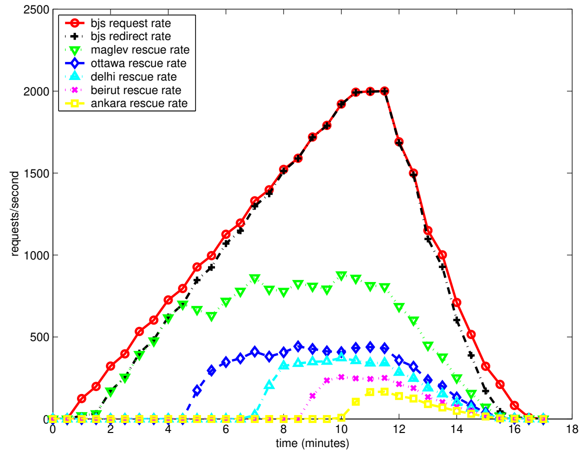


(b) Using the DotSlash rescue services

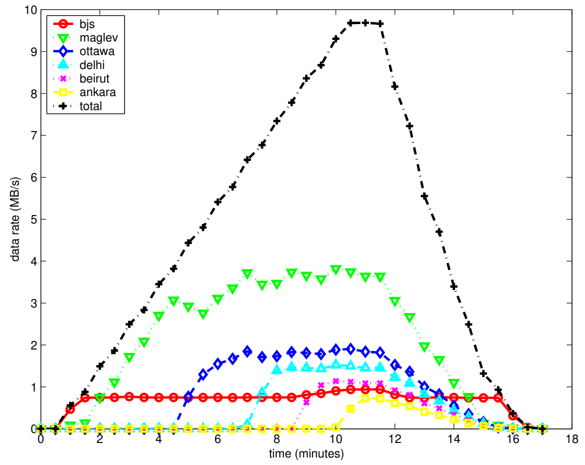
Figure 10: The data rate and request rate for a PlanetLab DSL node *gtidsl1* in two cases

6.3.4 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 the 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*



(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

Figure 11: The request rates and data rates at the origin server *bjs* and its rescue servers

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, six machines, *bjs*, *maglev*, *ottawa*, *delhi*, *beirut*, and *ankara*, run as web servers, where the first two are iDot machines, and the last four 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 D_{out} as 1000, 10000, 5000, 4000, 3000, and 2000 kB/s, respectively. We run httperf on five clic machines, which 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 six web servers at an interval of 30 seconds. The retrieved status data are stored in round-robin databases using RRDtool [28], with one database for each web server. Figure 11 shows the data rates and request rates for the six web servers in a duration of 17 minutes.

We observe the following results from Figure 11(a). First, *bjs* can support a request rate of 2000 requests/second. Its redirect rate increases as the request rate increases, and these two rates are roughly the same once the request rate exceeds 1500 requests/second. This validates the control mechanisms we described in Section 4.4.4: (1) the maximum rate of HTTP redirects *bjs* can support is bounded by $D_{out}/A_{redi} = 2140$ requests/second, and (2) if the rate of HTTP redirects is greater than $D_{out} * T_{high}/A_{redi} = 1603$ requests/second, $T_{redi} = 0$, which means that all client requests are redirected from *bjs* to its rescue servers. Second, *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 Figure 11(b) and 11(a), we observe that rescue servers have similar curve shapes for their data rates and rescue rates. In contrast, as we expected, the origin server *bjs* has quite different curve shapes for its data rate and request rate. Although the request rate at *bjs* increases significantly from 200 requests/second at 1.5 minutes to 2000 requests/second at 11 minutes, the data rate at *bjs* is roughly unchanged, staying at $D_{out} * T_{high} = 750$ 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 the data rate at *bjs* goes beyond 750 kB/s, but still stays below $D_{out} = 1000$ kB/s, when the request rate is between 1600 and 2000 requests/second. 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 existing web server infrastructure, such as CDNs, to handle web hotspots effectively. It is scalable, cost-effective, easy to use, self-configuring, and transparent to clients. Through our preliminary experimental results, we have demonstrated the advantages of using DotSlash, where a web server achieved an order of magnitude improvement for the request rate its supported and the data rate delivered to clients, 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.

8. REFERENCES

[1] Tarek F. Abdelzaher and Nina 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] Mohit Aron, Darren Sanders, Peter Druschel, and Willy 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, 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.

[9] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2):263–311, June 2002.

[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] Ethereal homepage. http://www.ethereal.com.

[12] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force, January 1997.

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[14] E. Guttman. Attribute list extension for the service location protocol. RFC 3059, Internet Engineering Task Force, February 2001.

[15] E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.

[16] Jaeyeon Jung, Balachander Krishnamurthy, and Michael 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.

[17] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *The 29th ACM Symposium on Theory of Computing (STOC)*, El Paso, Texas, May 1997.

[18] Keith Kong and Dipak Ghosal. Mitigating server-side congestion in the Internet through pseudoserving. *IEEE/ACM Transactions on Networking*, 7(4):530–544, August 1999.

[19] W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at USENIX LISA'01, December 2001.

[20] Quanzhong Li and Bongki Moon. Distributed cooperative Apache web server. In *International World Wide Web Conference*, Hong Kong, May 2001.

[21] David Mosberger and Tai Jin. httperf—a tool for measuring web server performance. In *Workshop on Internet Server Performance (WISP)*, Madison, Wisconsin, June 1998.

[22] Netcraft. Web server survey. http://news.netcraft.com/archives/web_server_survey.html.

[23] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.

[24] PlanetLab homepage. http://www.planet-lab.org/.

[25] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM Symposium on Communications Architectures and Protocols*, San Deigo, California, August 2001.

[26] J. Rosenberg, Henning Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001.

[28] RRDtool homepage. http://people.ee.ethz.ch/˜oetiker/webtools/rrdtool/.

[29] Stan Schwarz. Web servers, earthquakes, and the slashdot effect. http://pasadena.wr.usgs.gov/office/stans/slashdot.html.

[30] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer caching schemes to address flash crowds. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.

[31] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. A lightweight, robust P2P system to handle flash crowds. In *IEEE International Conference on Network Protocols (ICNP)*, Paris, France, November 2002.

[32] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM Symposium on Communications Architectures and Protocols*, San Deigo, California, August 2001.

[33] Universal description, discovery and integration (UDDI) homepage. http://www.uddi.org/.

[34] Universal plug and play (UPnP) homepage. http://www.upnp.org.

[35] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, December 2003.

[36] 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.

[37] Jim Waldo. The Jini architecture for network-centric computing. *Communications ACM*, 42(7):76–82, July 1999.

[38] Jia Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Review (CCR)*, 29(5), October 1999.

[39] Limin Wang, Vivek Pai, and Larry Peterson. The effectiveness of request redirection on CDN robustness. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, Massachusetts, December 2002.

[40] Matt Welsh and David Culler. Adaptive overload control for busy Internet servers. In *USENIX Conference on Internet Technologies and Systems (USITS)*, Seattle, Washington, March 2003.

[41] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, April 2001.

[42] W. Zhao, H. Schulzrinne, and E. Guttman. Mesh-enhanced service location protocol (mSLP). RFC 3528, Internet Engineering Task Force, April 2003.

[43] W. Zhao, Henning 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.

[44] Weibin Zhao and Henning Schulzrinne. Selective anti-entropy. In *ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.