Chi-Hung Chi
Maarten van Steen
Craig Wills (Eds.)

LNCS 3293

Web Content Caching and Distribution

9th International Workshop, WCW 2004
Beijing, China, October 2004
Proceedings

Springer

Chi-Hung Chi Maarten van Steen
Craig Wills (Eds.)

Web Content Caching and Distribution

9th International Workshop, WCW 2004
Beijing, China, October 18-20, 2004
Proceedings

Springer

Volume Editors

Chi-Hung Chi
National University of Singapore
School of Computing
Lower Kent Ridge Road, Singapore 119260
E-mail: chich@comp.nus.edu.sg

Maarten van Steen
Vrije Universiteit Amsterdam
Department of Computer Science
De Boelelaan 1081a, 1081 HC Amsterdam, The Netherlands
E-mail: steen@cs.vu.nl

Craig Wills
Worcester Polytechnic Institute
Computer Science Department
100 Institute Road, Worcester, MA 01609, USA
E-mail: cew@cs.wpi.edu

Library of Congress Control Number: 2004113293

CR Subject Classification (1998): C.2, H.4, D.1.3, D.4.4, H.5

ISSN 0302-9743
ISBN 3-540-23516-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 11335207 06/3142 5 4 3 2 1 0

Preface

Since the start of the International Workshop on Web Caching and Content Distribution (WCW) in 1996, it has served as the premiere meeting for researchers and practitioners to exchange results and visions on all aspects of content caching, distribution, and delivery. Building on the success of the previous WCW meetings, WCW 2004 extended its scope and covered interesting research and deployment areas relating to content services as they move through the Internet.

This year, WCW was held in Beijing, China. Although it was the first time that WCW was held in Asia, we received more than 50 high quality papers from five continents. Fifteen papers were accepted as regular papers and 6 papers as synopses to appear in the proceedings. The topics covered included architectural issues, routing and placement, caching in both traditional content delivery networks as well as in peer-to-peer systems, systems management and deployment, and performance evaluation.

We would like to take this opportunity to thank all those who submitted papers to WCW 2004 for their valued contribution to the workshop. This event would not have been possible without the broad and personal support and the invaluable suggestions and contributions of the members of the program committee and the steering committee.

August 2004

Chi-Hung Chi
Maarten van Steen
Craig Wills

General Chair

Chi-Hung Chi
Kwok-Yan Lam

Steering Committee

Azer Bestavros
Pei Cao
Jeff Chase
Brian Davison
Fred Douglis
Michael Rabinovich
Duane Wessels

Program Committee

Maarten van Steen (chair)
Craig Wills (vice-chair)
Gustavo Alonso
Chin-Chen Chang
Chi-Hung Chi
Michele Colajanni
Mike Dahlin
Magnus Karlsson
Ihor Kuz
Kwok-Yan Lam
Dan Li
Pablo Rodriguez
Oliver Spatscheck
Geoff Voelker
Limin Wang
Tao Wu
Zheng Zhang

Table of Contents

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 web server

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

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.

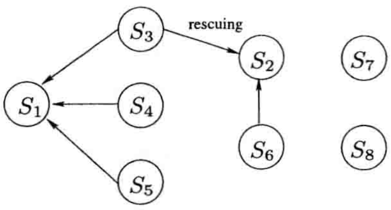


Fig. 1. An example for DotSlash rescue relationships

3 DotSlash Overview

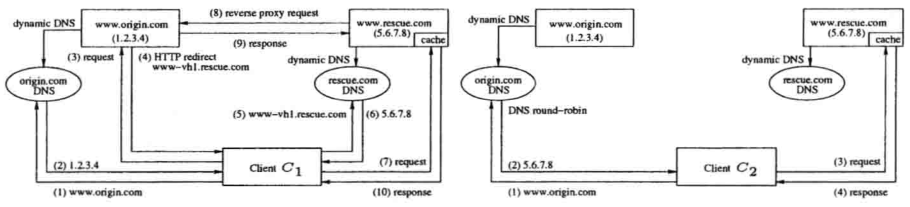
DotSlash uses a mutual-aid rescue model. A web server joins a mutual-aid community by registering itself with a DotSlash service registry, and contributing its spare capacity to the community. In case of being heavily loaded, a participating server discovers and uses spare capacities at other servers 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