DotSlash: Handling Web Hotspots at Dynamic Content Web Sites*

Weibin Zhao, Henning Schulzrinne
Department of Computer Science
Columbia University
New York, NY 10027
{zwb,hgs}@cs.columbia.edu

Abstract

We propose DotSlash, a self-configuring and scalable rescue system, for handling web hotspots at dynamic content web sites. To support load migration for dynamic content, an origin web server sets up needed rescue servers drafted from other web sites on the fly, and those rescue servers retrieve the scripts dynamically from the origin web server, cache the scripts locally, and access the corresponding database server directly. We have implemented a prototype of DotSlash for the LAMP configuration, and tested our implementation using the RUBBoS bulletin board benchmark. Experiments show that by using DotSlash a dynamic content web site can completely remove its web server bottleneck, and can support a request rate constrained only by the capacity of its database server.

1 Introduction

Handling web hotspots (also known as flash crowds or the Slashdot effect [1]) at dynamic content web sites is a challenging problem. First, a dynamic content web site is more likely to be overwhelmed by flash crowds because the request rate it supports is often much lower than that of a static content web site [9] since generating dynamic web pages consumes more CPU cycles than serving static web pages. Secondly, many existing caching mechanisms are designed for static content, and cannot be applied to dynamic content directly [9, 10, 20]. Furthermore, a dynamic content web site typically has a three-tier architecture as illustrated in Figure 1: a front-end web server handles the HTTP requests from clients, an application server implements the business logic, and a back-end database server stores the content. Depending on different applications and system configurations, different servers in the infrastructure may become the bottleneck [4, 8].

In our earlier work [21], we have developed DotSlash, a self-configuring and scalable rescue system for handling web hotspots for static content effectively.

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

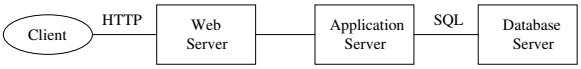

Figure 1: The three-tier architecture for dynamic content web sites

Here, we extend the DotSlash framework to support web hotspot rescue for dynamic content.

Dynamic content can be generated using different technologies, such as PHP, Active Server Pages (ASP), Java Server Pages (JSP), Java Servlets, and Enterprise Java Beans (EJB). PHP is the most popular dynamic web technology used with Apache, and Apache is the most popular web server. Thus, we discuss DotSlash in the context of the widely used LAMP configuration (Linux, Apache, MySQL, and PHP), and expect that similar techniques can be applied to other configurations of dynamic content web sites [8, 18]. In the LAMP configuration as shown in Figure 2, PHP is a module of the Apache web server, and the web server and database server are usually running on separate machines.

Amza et al. [4] show that different applications may have different bottlenecks in the LAMP configuration. The database server is the bottleneck for the TPC-W benchmark [19] that models online bookstores such as amazon.com. But the web server is the bottleneck for the RUBiS benchmark [15] that models auction sites such as eBay, and for the RUBBoS benchmark [16] that models bulletin board sites such as Slashdot. We focus on the web server bottleneck in this paper, and will address the database server bottleneck in the next stage of this project. Our approach is as follows. When a web server is heavily loaded, it drafts a number of rescue servers from other web sites on the fly, and redirects a fraction of client requests to those rescue servers. To serve redirected client requests, a rescue server retrieves the PHP scripts dynamically from its origin server, caches the scripts locally, and accesses the corresponding database server directly. We have implemented a prototype of DotSlash for the LAMP configuration, and tested our implementation using the RUBBoS bulletin board benchmark [4]. Experiments show

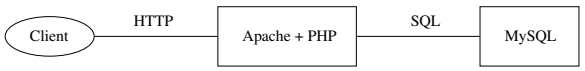

Figure 2: The LAMP configuration for dynamic content web sites

that by using DotSlash a dynamic content web site can completely remove its web server bottleneck, and can support a request rate constrained only by the capacity of its database server.

The remainder of this paper is organized as follows. We discuss related work in Section 2, give an overview of the DotSlash architecture in Section 3, and describe dynamic script replication in Section 4. We present experimental results and evaluations in Section 5, and conclude in Section 6.

2 Related Work

Various approaches have been proposed to cache dynamic content. Web caching can cache entire HTML pages or page fragments at proxies [10], web servers [9], application servers [11, 5], and edge servers [2]. Database caching [3, 6, 12] can cache data from the back-end database at database caches closer to the application server. Complementary to existing caching mechanisms, DotSlash allows a web site to expand its capacity dynamically as load increases without administrator interventions. In particular, DotSlash allows a web server to obtain additional computing capacity on demand and replicate scripts dynamically.

In edge computing [2], application components can be offloaded to edge servers, but manual configuration is needed to choose the components to be offloaded and where to deploy applications. In ACDN [13], applications can be deployed and re-deployed dynamically, but manual administration is still involved such as creating a meta-file for each application to be replicated. In contrast, DotSlash is self-managing by replicating each script file on demand and fully automatically.

To address the database bottleneck, various schemes for database caching and replication [3, 6, 12, 17, 14] have been proposed. We plan to incorporate effective mechanisms into DotSlash to automate data replication for web hotspot rescue in the future.

3 The DotSlash Architecture

We provide a brief overview of the DotSlash architecture here; a more complete description is given in [21].

DotSlash uses a mutual-aid rescue model, where different web sites form a mutual-aid community and use spare capacity in the community to relieve web hotspots experienced by any individual site. DotSlash consists of service discovery, workload monitoring, request redirection, dynamic virtual hosting, and rescue control. DotSlash allows a web site to build an adaptive distributed web server system in a fully automated way: service discovery [22] enables servers of different web sites to learn about each other dynamically and collaborate without any administrator intervention; dynamic virtual hosting enables a rescue server to serve the content of its origin servers on the fly; and rescue control allows a web server to tune its resource utilization by using rescue actions triggered automatically based on load conditions.

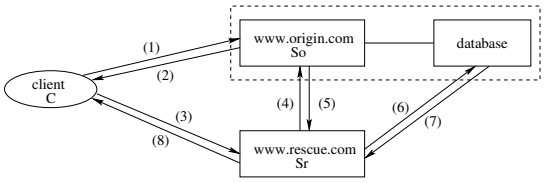

Figure 3: An example for dynamic script replication

4 Dynamic Script Replication

To support load migration for dynamic content, we enhance DotSlash with dynamic script replication, which allows a rescue server to dynamically replicate scripts from its origin servers, and cache the scripts locally. The motivation is that running scripts consumes a fair amount of CPU cycles, and the CPU often becomes the bottleneck for dynamic content web sites [4].

In DotSlash, an origin web server uses both DNS round robin and HTTP redirect to offload a fraction of client requests to its rescue servers [21]. For clarity, we omit DNS name resolution steps, and give an example that uses HTTP redirect to illustrate how dynamic script replication works. In Figure 3, the origin server S_o is *www.origin.com*, and the rescue server S_r is *www.rescue.com*. The client C takes the following steps to retrieve *http://www.origin.com/search.php?name=x*:

1. C makes an HTTP request to S_o using *http://www.origin.com/search.php?name=x*.
2. S_o sends an HTTP redirect to C as *http://www-vh1.rescue.com/search.php?name=x*. Note that S_o has already set up a rescue relationship with S_r.
3. C makes an HTTP request to S_r using *http://www-vh1.rescue.com/search.php?name=x*.
4. S_r makes an HTTP request to S_o using *http://www.origin.com/search.php* because of a cache miss for the script file *search.php*.
5. S_o sends the script file *search.php* to S_r, and S_r caches *search.php* locally.
6. S_r runs *search.php?name=x* to access the corresponding database.
7. S_r gets the query results from the database.
8. S_r sends the query results to C.

4.1 Operations at the Rescue Server

When the rescue server S_r receives a request Q, it checks whether Q is a redirected request for dynamic content. If Q uses an alias of S_r such as *www-vh1.rescue.com*, or uses an origin server's name such as *www.origin.com*, then Q is a redirected request. If the requested file extension matches one of the configured script extensions, then Q requests dynamic content. In Apache, script extensions are configured using the directive *AddType*, e.g., files with an extension of *php* or *phtml* can be configured as PHP scripts using "*AddType application/x-httpd-php .php .phtml*".

If Q is a redirected request for dynamic content such as *http://www-vh1.rescue.com/search.php?name=x*, then S_r maps Q's URI to a script file, and sets the needed environment variables for retrieving PHP scripts if there is a cache miss. In Apache, environment variables for sub-processes are set in a per request table *subprocess_env*. DotSlash sets three environment variables: *Origin_Server* which specifies the origin server's name, *Origin_Port* which specifies the origin server's port number for web requests, and *Script_Root* which specifies the root directory for replicated scripts. Since S_r may need to retrieve and cache scripts from multiple origin servers, a request URI is mapped to its script file as *Script_Root/Origin_Server/URI_Path*, where *URI_Path* is the path part of the request URI. For example, Q's URI is mapped to a script file Q_f as *Script_Root/www.origin.com/search.php*.

If Q_f exists, the script will be executed normally; otherwise, a "file not found" error will be triggered, and be handled by a 404 handler as follows. If *Script_Root* is set (i.e., a redirected request for dynamic content), the DotSlash inclusion function *dots_include* is invoked; otherwise, a regular "file not found" message is returned. Dynamic script replication is performed by *dots_include* using the following steps.

1. Retrieve the script file from S_o using *http://Origin_Server:Origin_Port/URI_Path*;
2. Add a header H to the retrieved script file for handling file inclusions (see Section 4.3 for details);
3. Set query variables (extracted from the query part of the request URI) in *$_GET* or *$_POST*;
4. Run the script by invoking the native PHP *include*.

File locking is used to ensure that partially retrieved script files are not used by concurrent requests.

4.2 Operations at the Origin Server

When the origin server S_o receives a request Q, it checks whether Q is from a rescue server (based on its rescue server list, see [21] for details), and whether Q is for dynamic content. If so, S_o will return the script file to the rescue server instead of running the script.

4.3 File Inclusions in Replicated Scripts

In PHP, file inclusions are supported via *include*, *require*, *include_once*, and *require_once* statements. A challenging issue here is that a replicated script running at a rescue server may include files located at the origin server.

We investigated two options for handling file inclusions in replicated scripts: renaming and error handler. The renaming approach is to rename each PHP inclusion statement to the DotSlash inclusion function *dots_include* after a script is replicated from the origin server to the rescue server. This approach is applicable to all PHP inclusion statements, but it needs to parse each replicated script file. The error handler approach is to use a customized error handler for each replicated script file. In PHP, if a file to be included does not exist, an error will be triggered. Thus, we can use a customized error handler to catch file inclusion errors, and replicate needed script files dynamically.

We employ the error handler approach in DotSlash mainly because it is easier to build. We add a header H to each replicated script file, which uses *set_error_handler* to set the error handler to the DotSlash error handler *dots_error*. As a wrapper function of *dots_include*, *dots_error* implements the PHP error handler API, and invokes *dots_include* in case of a file inclusion error.

4.4 Implementation

DotSlash functions, *dots_include* and *dots_error*, can be implemented as PHP user functions written in PHP scripts, or as PHP native functions written in C and compiled as the DotSlash extension to the PHP module. For efficiency considerations, we implement *dots_include* and *dots_error* as PHP native functions.

5 Evaluation

We use R to denote the maximum request rate supported by a web server. Our goal is to improve a web server's R by using DotSlash.

5.1 Experimental Setup

We have performed experiments in our local area networks, where we use a cluster of 30 Linux machines connected via 100 Mb/s fast Ethernet. These machines have two different configurations. The low-end configuration (*LC*) has a 1 GHz Intel Pentium III CPU, and 512 MB of memory, whereas the high-end configuration (*HC*) 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.

We run a varying number of web servers in different experiments. All web servers run Apache 2.0.49, configured with PHP 4.3.6, *worker* multi-processing module, proxy modules, cache modules, and our DotSlash

module. The PHP module includes our DotSlash extension, which implements the DotSlash inclusion function *dots_include* and the DotSlash error handler *dots_error*. In all experiments, we run one database server on an *HC* machine denoted as *DB_HC*. The database server runs MySQL 4.0.18. To support a large number of concurrent connections, we configure MySQL with *open_files_limit=65535*, and *max_connections=2048*.

We test our prototype system using the RUBBoS benchmark [4]. It has 19 PHP scripts, and the size of script files varies between 1 KB and 7 KB. The database has a size of 439 MB, and contains $500,000$ users and 2 years of stories and comments. There are 15 to 25 stories per day, and 20 to 50 comments per story. The length of story and comment bodies is between 1 KB and 8 KB.

We use RUBBoS clients to generate workloads. Each RUBBoS client can simulate a few hundred HTTP clients. An HTTP client issues a sequence of requests using a think time that follows a negative exponential distribution, with an average of 7 seconds [19]. If the request rate to be generated is high, multiple RUBBoS clients are used, each running on a separate machine. We use 7 seconds [7] as the timeout value for getting the response for a request. If more than 10% [7] of issued requests time out, the web server is considered overloaded.

5.2 Effectiveness

To show the effectiveness of DotSlash, we measure R at an origin web server from the client side in different cases, based on whether DotSlash is used or not, and whether the origin server runs on an *HC* machine or on an *LC* machine.

In the first experiment, we run the origin web server on an *HC* machine denoted as *Orig_HC*, and DotSlash is disabled. Figure 4 shows the experimental results. We denote the total number of HTTP clients as N_c. The request rate at *Orig_HC* increases as N_c increases. The measured R is 118 requests/second obtained when $N_c = 900$. When N_c reaches 1100, 11% of requests time out. At this workload, the CPU utilizations of *Orig_HC* and *DB_HC* are 100% and 45%, respectively. Clearly, the web server is the bottleneck, although it has the same hardware configuration as the database server.

In the second experiment, the origin web server still runs on *Orig_HC*, but DotSlash is enabled, and rescue servers are added automatically as load increases. All rescue web servers run on *LC* machines. We also show the experimental results in Figure 4, but for $N_c \geq 500$ only since *Orig_HC* does not use any rescue server when $N_c \leq 400$. The measured R is 245 requests/second obtained when $N_c = 1900$, and *Orig_HC* uses 9 rescue servers. When N_c reaches 2200, the database server *DB_HC* gets overloaded, where 16% of requests time out, and *Orig_HC* uses 10 rescue servers. At this workload, the CPU utilizations of *Orig_HC* and *DB_HC* are 60% and 100%, respectively, and the CPU utilizations of all rescue servers are below 60%.

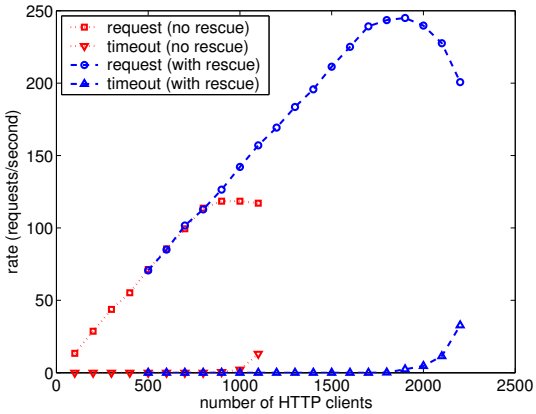

Figure 4: The request rate and timeout rate for the origin web server *Orig_HC* in two cases, namely without using DotSlash verses using DotSlash

Comparing the above two experiments, we have two results. First, in terms of the R supported by *Orig_HC*, we have $245/118 > 2$, meaning that we doubled the performance by using DotSlash. Secondly, based on the CPU utilization, we can observe that when DotSlash is used, the origin web server is no longer a bottleneck, and the performance is constrained only by the database server. To further verify this observation, we repeat the above two experiments by running the origin web server on an *LC* machine denoted as *Orig_LC* so that we have a low-end origin web server and a high-end database server. To save space, we summarize the experimental results as follows without showing figures.

Without using DotSlash, the measured R is 49 requests/second obtained when $N_c = 500$. *Orig_LC* gets overloaded when N_c reaches 600, where 21% of requests time out. When DotSlash is used, the measured R is 245 requests/second obtained when $N_c = 1900$, and *Orig_LC* uses 10 rescue servers. *DB_HC* gets overloaded when N_c reaches 2200, where 16% of requests time out, and *Orig_LC* uses 12 rescue servers. Thus, we have $245/49 = 5$, meaning that we improved the R at *Orig_LC* by 500% by using DotSlash. The reason for using 10 rescue servers to get this improvement is that the origin server and rescue servers have a CPU utilization close to 50% because we have configured the desired load region in our experiments as $[45\%, 70\%]$. More specifically, *Orig_LC* can support a rate of 49 requests/second with 100% CPU utilization. Thus, to support a rate of 245 requests/second, we need 5 such web servers with 100% CPU utilization, or equivalently, we can use 11 such web servers (i.e., 1 origin server and 10

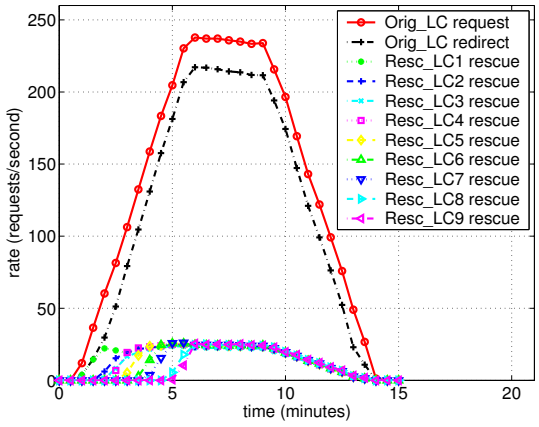

Figure 5: The request rate and redirect rate at the origin server *Orig_LC* and the rescue rate at the 9 rescue servers (*Resc_LC1*, ..., *Resc_LC9*)

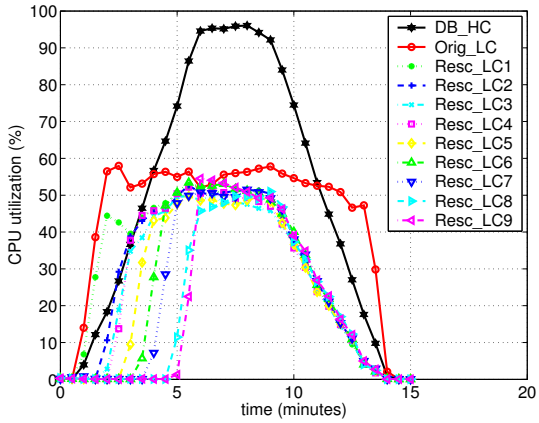

Figure 6: The CPU utilization for the origin server *Orig_LC*, the 9 rescue servers (*Resc_LC1*, ..., *Resc_LC9*), and the database server *DB_HC*

rescue servers) with $5 * 100\%/11 = 45\%$ CPU utilization.

From the above experiments, we can observe that using DotSlash can completely remove the web server bottleneck, and the performance of a dynamic content web site is constrained only by its database server. Also, when DotSlash is used, it does not make much difference as to using a high-end web server or a low-end web server. For example, to support a rate of 245 requests/second, *Orig_HC* uses 9 rescue servers whereas *Orig_LC* uses 10 rescue servers.

5.3 Workload Control and Migration

DotSlash monitors workload by maintaining a number of counters for outbound HTTP traffic and CPU utilization, and allows these counter values to be retrieved conveniently via *http://host.domain/dotslash-status?auto*. By sampling these counters at a desired interval, we can calculate the needed average values of request rate, redirect rate, rescue rate, and CPU utilization.

To show how workload is controlled and migrated at the server side, we perform the following experiment. The origin web server runs on *Orig_LC*, and all rescue web servers run on *LC* machines, denoted as *Resc_LC1*, ..., *Resc_LCn*. DotSlash is enabled, and rescue servers are added automatically as load increases. We run 5 RUBBoS clients, all using the same workload profile to issue requests to *Orig_LC*. Each RUBBoS client simulates 340 HTTP clients, thus a total of 1700 HTTP clients are simulated. We start one RUBBoS client at a time, with an interval of 1 minute, and each RUBBoS client runs for 8 minutes. We run a shell script to get the DotSlash status from all servers at an interval of 30 seconds. Figure 5 shows the request rate and redirect rate at *Orig_LC* and the rescue rate at the 9 rescue servers in a duration of 15 minutes. We can observe the following results. First, the redirect rate at *Orig_LC* increases as the request rate increases, meaning that excess workload is migrated from *Orig_LC* to its rescue servers via redirects. Secondly, the serving rate (i.e., the request rate minus the redirect rate) at *Orig_LC* decreases as the redirect rate increases because redirects consume CPU cycles. Also, the serving rate should be 22–34 requests/second for the desired CPU load region of $[45\%, 70\%]$ and a capacity rate of 49 requests/second, but the real serving rate is bit smaller, 20–30 requests/second, due to the redirect overhead. Finally, the rescue rate at all 9 rescue servers is about 25 requests/second, which is the workload that drives the CPU utilization to about 50% at rescue servers.

Figure 6 shows the CPU utilization for *Orig_LC*, the 9 rescue servers, and *DB_HC*. We can observe the following results. First, *Orig_LC* has successfully controlled its CPU utilization to stay within 50–60%. Secondly, all rescue servers have a CPU utilization of 45–55%, being close to 50% mostly. Finally, when N_c reaches 1700, *DB_HC* has a CPU utilization around 95%, meaning that without relieving the database server bottleneck, there is not much potential to further increase the request rate.

6 Conclusions

In this paper, we have described DotSlash, a new system for performing web hotspot rescue at dynamic content web sites. By supporting dynamic script replication, DotSlash can completely remove the web server bottleneck for dynamic content web sites. In future work, we plan to investigate mechanisms to relieve the database server bottleneck for web hotspot rescue.

References

[1] Stephen Adler. The slashdot effect: An analysis of three Internet publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.

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

[3] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In *International Conference on Data Engineering (ICDE)*, Bangalore, India, March 2003.

[4] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, Colorado, August 2002.

[5] BEA WebLogic. http://www.bea.com/products/weblogic/server/.

[6] C. Bornhovd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering Bulletin*, 27(2):11–18, June 2004.

[7] 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.

[8] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Middleware*, Rio de Janeiro, Brazil, June 2003.

[9] J. Challenger, P. Dantzig, A. Iyengar, M. Squillante, and L. Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 12(2):233–246, April 2004.

[10] A. Datta, K. Dutta, H. Thomas, D. Vandermeer, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation. *ACM Transactions on Database Systems*, 29(2):403–443, June 2004.

[11] IBM WebSphere. http://www-306.ibm.com/software/websphere/.

[12] P. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL server. *Data Engineering Bulletin*, 27(2):35–40, June 2004.

[13] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating Internet applications. In *International Workshop on Web Caching and Content Distribution (WCW)*, Hawthorne, NY, September 2003.

[14] S. Ranjan, R. Karrer, and E. Knightly. Wide area redirection of dynamic content by Internet data center. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Hong Kong, China, March 2004.

[15] Rice university bidding system. http://www.cs.rice.edu/CS/Systems/DynaServer/RUBiS/.

[16] Rice university bulletin board system. http://www.cs.rice.edu/CS/Systems/DynaServer/RUBBoS/.

[17] S. Sivasubramanian, G. Pierre, and M. Steen. Replicating web applications on-demand. In *International Conference on Services Computing (SCC)*, Shanghai, China, September 2004.

[18] L. Titchkosky, M. Arlitt, and C. Williamson. A performance comparison of dynamic web technologies. *ACM SIGMETRICS Performance Evaluation Review*, 31(3):2–11, December 2003.

[19] Transaction processing performance council. http://www.tpc.org/tpcw/.

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

[21] W. Zhao and H. Schulzrinne. DotSlash: A self-configuring and scalable rescue system for handling web hotspots effectively. In *International Workshop on Web Caching and Content Distribution (WCW)*, Beijing, China, October 2004.

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