DotSlash: An Automated Web Hotspot Rescue System with On-demand Query Result Caching†

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

***Abstract*— DotSlash is an automated web hotspot rescue system. This paper presents DotSlash *Qcache* services that allow a web site to use on-demand distributed query result caching to greatly reduce the workload at read-mostly databases. DotSlash Qcache services complement DotSlash rescue services; together they provide a comprehensive solution to address different bottlenecks at multi-tier web sites.**

I. Introduction

Web hotspots, also known as flash crowds or the Slashdot effect [1], are short-term dramatic load spikes that could seriously degrade the service quality of affected web sites. To handle web hotspots cost-effectively, we developed DotSlash, a self-configuring and scalable rescue system [22], [23].

This paper presents DotSlash *Qcache* services, which allow a web site to use on-demand distributed query result caching to greatly reduce the workload at read-mostly databases. The novelty of this work is on-demand caching based on load conditions: caching remains inactive as long as the load is normal, but is activated once the load is heavy. This approach offers good data consistency for normal load and good scalability with relaxed data consistency under heavy load. DotSlash Qcache services complement DotSlash rescue services; together they provide a comprehensive solution to address different bottlenecks at multi-tier web sites.

The remainder of this paper is organized as follows. We discuss related work in Section II, introduce DotSlash system architecture in Section III, describe the design of DotSlash Qcache services in Section IV, evaluate our prototype system in Section V, and conclude in Section VI.

II. Related Work

A large body of existing work on web hotspots [8], [19], [14] have focused on static content. To improve the application server scalability, application programs or components [16], [2] can be offloaded from the origin server. Recently, Olston et al. [12] proposed a scalability service for databases using multicast-based consistency management. Although their system aims at broader web applications, its scalability gain under heavy load is unclear. In contrast, our system targets read-mostly databases and scales well under dramatic load spikes.

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

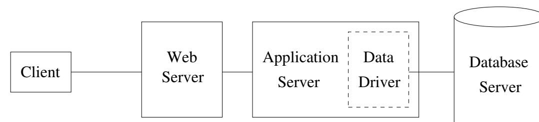

Fig. 1. DotSlash Application Model

Caching is very effective for web content distributions. Web caching [7], [6] can cache HTML pages or page fragments, whereas database caching [3], [5] can cache data from back-end databases. While caching in existing systems is active in all cases, our query result caching is activated only under heavy load, which minimizes the effect of caching on consistency while improving the system scalability.

Replication [15], [18] is a widely used mechanism for database scalability. Our current prototype uses a single back-end database server, which can be extended to support distributed database servers by incorporating database replication into our system.

Database clustering [13], [11] is a mechanism at the database server tier to pool database servers together for high availability and performance. DotSlash is a solution at the web/application server tier for scalability. Our system and database clustering are orthogonal, and they can be used together at dynamic content web sites.

III. DotSlash System Architecture

The application model for our system is the standard three-tier web architecture as shown in Figure 1, where application programs running at the application server access application data stored in the database server through a data driver.

A. DotSlash Usage Models

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. We consider three types of mutual-aid communities: open communities that are intended for a cooperative environment, closed communities that use authentication to only admit authorized web sites, and flood-insurance closed communities that employ tokens as well as authentication to promote the incentive for providing DotSlash rescue services and reduce abuse.

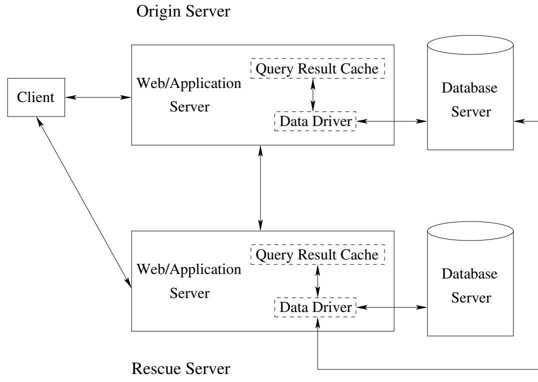

Fig. 2. Enabling query result caching in DotSlash

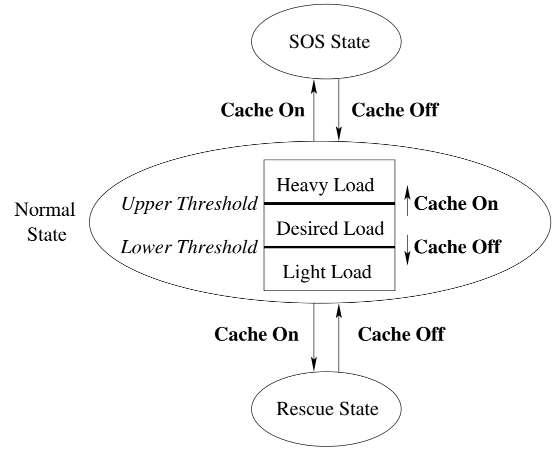

Fig. 3. DotSlash on-demand query result caching

B. DotSlash Rescue Services

DotSlash rescue services enable a web site to build an adaptive distributed web server system on the fly and replicate application programs dynamically, which relieve a spectrum of bottlenecks ranging from access network bandwidth to web servers and application servers. An origin web server discovers suitable rescue servers via wide-area service location, allocates them for temporary use, and redirects client requests to them. DotSlash uses DNS round robin as the first level crude load distribution, and uses HTTP redirect as the second level fine-grained load balancing. When a rescue relationship is set up, the rescue server assigns a unique virtual host name to the origin server, which is used by the origin server in its HTTP redirects to the rescue server. Also, the origin server adds the rescue server's IP address to its local DNS for round robin. A rescue server works as a reverse caching proxy for its origin server, and serves the content of its origin server on the fly.

C. DotSlash Qcache Services

DotSlash Qcache services allow an origin server and its rescue servers to use on-demand query result caching to reduce the database workload at the origin server. Since the data driver intercepts all database queries, we enhance it with query result caching without changing the application API and database interface. In our prototype system [20], we use the common LAMP (Linux, Apache, MySQL, and PHP) configuration, and extend the original PHP data driver for MySQL databases with a query result cache. Figure 2 illustrates how to enable query result caching in DotSlash. Note that a client request can be redirected from the origin server to the rescue server via either DNS round robin or HTTP redirect.

IV. The Design of DotSlash Qcache Services

A. Caching Features

Our query result caching is on-demand, self-configuring, distributed, and transparent to web users and applications.

The control of our on-demand caching is based on two factors, namely the web server's DotSlash state and load region. A web server has three DotSlash states: *SOS state* if it gets rescue services from others, *rescue state* if it provides rescue services to others, and *normal state* otherwise. DotSlash uses two configurable parameters, lower threshold ρ^l and upper threshold ρ^u, to define three load regions: light load region $[0, \rho^l)$, desired load region $[\rho^l, \rho^u]$, and heavy load region $(\rho^u, 100\%]$. Our current prototype system measures network and CPU utilization.

Figure 3 shows the control of our on-demand caching. Caching is activated if a web server is in the SOS state (i.e., an origin server), or if a web server is in the rescue state (i.e., a rescue server), or if a web server is in the normal state and its load is above the upper threshold. On the other hand, caching is de-activated when an origin server switches from the SOS state to the normal state, or when a rescue server switches from the rescue state to the normal state, or when a web server is in the normal state and its load is below the lower threshold.

When an origin server sets up its rescue servers, it passes the query result caching control parameters to its rescue servers. By doing so, a rescue server can manage cached objects based on the instructions from the origin server. In this way, an origin server can set up a distributed query result caching system on the fly using one set of control parameters.

By default, each web/application server has its own, co-located query result cache. An origin server can obtain more query result caches as it drafts more rescue servers. Using co-located query result caches is well-suited for DotSlash in terms of resource utilization efficiency because our query result caching is on demand, and the cache server is idle most of time. Note that our system can use a dedicated query result cache server which is shared among an origin server and its rescue servers, or among a subset of rescue servers. Doing so can reduce the workload at the origin database server. However, a shared cache may become a potential performance bottleneck, and accessing a remote cache incurs longer delays.

Without the need to change client-side web browsers and server-side application programs, our query result caching is easy to deploy. Furthermore, we provide a way for web users to bypass our query result caching. Our current prototype

TABLE I
DOTSLASH CACHING-ENHANCED DATA DRIVER

Cache On	Bypass Cache	Rescue Request	Database Write	Database Read
no	–	–	normal	normal
yes	no	–	turn off	cache+DB
yes	yes	no	normal	DB+cache
yes	yes	yes	redirect	redirect

system uses the HTTP *Cache-Control* header for this purpose as follows. If there is *no-cache* or *max-age=0* in the HTTP Cache-Control header of a client request, DotSlash will handle that request without using query result caching.

B. Caching-enhanced Data Driver

Our caching-enhanced data driver is designed with the following considerations. First, rescue servers only handle read-only queries; all write queries are handled by the origin web server. This is mainly for security reasons because an origin server is unlikely to allow rescue servers to update its databases. Secondly, under heavy load we turn off write queries temporarily for regular users, but still allow site administrators to perform necessary updates. This is mainly for scalability considerations because a large number of read/write contentions can seriously degrade the database performance. We use an application-specific caching TTL to bound the staleness of cached objects. Note that this design targets hotspot rescue for read-mostly databases, which are common for content management systems (CMS), blogs, and web forums. It does not aim to be applicable to all web applications.

Our caching-enhanced data driver handles database queries based on three factors, namely the web server's query result caching state, the client request HTTP Cache-Control header, and the client request type. For a client request, if its HTTP *Host* header uses an origin server's name such as *www.origin.com*, or an assigned virtual host name such as *vh1.www.rescue.com*, then the request type is *rescue*; otherwise the request type is *regular*.

Table I shows the control of our caching-enhanced data driver. There are four cases. For case 1, query result caching is off. Any database query is handled normally by forwarding the query directly to the database. For case 2, query result caching is on and bypassing caching is false. Any write query is turned off, and an error message is returned. Any read-only query is handled as follows. Check the query against the query result cache. For a cache hit, get the result from the cache, and return it to the application. For a cache miss, submit the query to the corresponding database, which can be a local database or a remote database at the origin server; then get the query result from the database, save it to the query result cache, and return it to the application. For case 3, query result caching is on, bypassing caching is true, and the request type is regular. Any database query is forwarded directly to the database. For a read-only query, save the query result to the query result cache before returning it to the application. For case 4, query result caching is on, bypassing caching is true, and the request type is rescue. The request is redirected back to the origin web server via HTTP redirect, which is to ensure that a client request that needs to bypass caching can always be handled by the origin web server. For this purpose, an origin server does not apply HTTP redirect to client requests that need to bypass caching. However, client requests could be distributed to rescue servers due to the origin server's DNS round robin. This is why we need to use HTTP redirect in case 4. Note that a rescue server uses the origin server's IP address in its HTTP redirects to bypass the origin server's DNS round robin.

C. Query Result Cache

Both disk and memory can be used as our caching storage engine. Due to performance considerations, we choose to use a memory storage engine called *memcached* [10], which employs a client-server model. At the server side, a daemon maintains cached objects in dynamically allocated memory. Each cached object is a key-value pair with an expiration time. At the client side, we use an open-source C library *libmemcache* [9] to access the cache. In the *check_in* function, we first use the ELF hash algorithm [4] to map the query string into a cache key, and then store the query string and the query result as the cache value, using the caching TTL as the expiration time. In the *check_out* function, we use the same ELF hash algorithm to map the query string into a cache key. If a cached object is found for the key, we check whether the stored query string matches the input query string. If so, it is a cache hit; otherwise, it is a cache miss.

V. EVALUATION

We evaluate DotSlash rescue services and Qcache services individually as well as altogether using the RUBBoS bulletin board benchmark [17]. We carry out experiments in our local area network using a cluster of Linux machines connected via 100 Mb/s fast Ethernet. All web/application servers run Apache 2.0.49, configured with PHP 4.3.6. The database server runs MySQL 4.0.18. We use our *dot-slash.net* domain for dynamic DNS updates, and use the enhanced Service Location Protocol [21] for rescue server discovery.

A. Results for Read-only Mix

We first test our system using the RUBBoS read-only mix. Depending on whether rescue servers are available, whether query result caching is enabled, and whether each web/application server has a co-located cache or uses a shared cache server running on a separate machine, we have five test cases as follows.

- $READ$: no rescue, no cache.
- $READ_c$: no rescue, with a co-located cache.
- $READ_r$: with rescue, no cache.
- $READ_{r,c}$: with rescue, with a co-located cache.
- $READ_{r,sc}$: with rescue, with a shared cache.

Table II summaries the performance of our prototype system for the RUBBoS read-only mix. Without using DotSlash

TABLE II

PERFORMANCE SUMMARY FOR RUBBOS READ-ONLY MIX

Test Case	Max Rate (reqs/s)	Compared to $READ$	Compared to $READ_r$	Rescue Servers
$READ$	117	100%		
$READ_c$	125	107%		
$READ_r$	249	213%	100%	4
$READ_{r,c}$	1151	984%	462%	15
$READ_{r,sc}$	828	708%	333%	13

rescue and Qcache services, a web server can only support a request rate of 117 requests/second. The request rate supported increases to 249 requests/second by only using DotSlash rescue services with 4 rescue servers, and increases to 1151 requests/second by using DotSlash rescue and Qcache services together with 15 rescue servers. Compared to $READ$, $READ_r$ and $READ_{r,c}$ achieve an improvement of 213% and 984%, respectively. Compared to $READ_r$, $READ_{r,c}$ achieves an improvement of 462%.

B. Results for Submission Mix

Based on Section IV-B, DotSlash turns off database write queries temporarily for regular users under heavy load. We disable this feature in testing our system against the RUBBoS submission mix, which has about 2% write queries. We choose to do so for two reasons. First, turning off all write queries will convert the submission mix into a read-only mix. Secondly, allowing site administrators to perform necessary updates in our system is roughly equivalent to having a small percentage of write queries in the submission mix. Depending on whether rescue servers are available and whether query result caching is enabled, we have four test cases as follows.

- SUB: no rescue, no cache.
- SUB_c: no rescue, with cache.
- SUB_r: with rescue, no cache.
- $SUB_{r,c}$: with rescue, with cache.

Table III summarizes the performance of our prototype system for the RUBBoS submission mix. Without using DotSlash rescue and Qcache services, a web server can only support a request rate of 180 requests/second. The request rate supported increases to 580 requests/second by only using DotSlash rescue services with 4 rescue servers, and increases to 871 requests/second by using DotSlash rescue and Qcache services together with 8 rescue servers. Compared to SUB, SUB_r and $SUB_{r,c}$ achieve an improvement of 322% and 484%, respectively. Compared to SUB_r, $SUB_{r,c}$ achieves an improvement of 150%.

VI. CONCLUSIONS

In this paper, we have described how to enable on-demand distributed query result caching in DotSlash for handling web hotspots effectively. Through our experimental results, we have demonstrated that using DotSlash rescue and Qcache services together is very effective for read-mostly databases, e.g., a web site can improve its maximum request rate supported by a factor of 10 for the RUBBoS read-only mix.

TABLE III

PERFORMANCE SUMMARY FOR RUBBOS SUBMISSION MIX

Test Case	Max Rate (reqs/s)	Compared to SUB	Compared to SUB_r	Rescue Servers
SUB	180	100%		
SUB_c	174	97%		
SUB_r	580	322%	100%	4
$SUB_{r,c}$	871	484%	150%	8

REFERENCES

[1] S. 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] Andrew Binstock. Hashing rehashed. Dr. Dobb's, April 1996.

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

[6] 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.

[7] 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.

[8] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with Coral. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, March 2004.

[9] libmemcache homepage. http://people.freebsd.org/~seanc/libmemcache/.

[10] memcached homepage. http://www.danga.com/memcached/.

[11] MySQL cluster. http://www.mysql.com/products/database/cluster/.

[12] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *The Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 2005.

[13] Oracle real application clusters (RAC). http://www.oracle.com/technology/products/database/clustering/index.html.

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

[15] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *ACM/IFIP/USENIX International Middleware Conference*, Toronto, Canada, October 2004.

[16] 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.

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

[18] S. Sivasubramanian, G. Alonso, G. Pierre, and M. v. Steen. GlobeDB: Autonomic data replication for web applications. In *International World Wide Web Conference (WWW)*, Chiba, Japan, May 2005.

[19] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Annual Usenix Technical Conference*, Boston, Massachusetts, June 2004.

[20] Weibin Zhao and Henning Schulzrinne. DotSlash—an automated web hotspot rescue system project. http://dotslash.sourceforge.net/.

[21] Weibin Zhao and Henning Schulzrinne. Service location protocol enhancements project. http://mslp.sourceforge.net/.

[22] Weibin Zhao and Henning 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.

[23] Weibin Zhao and Henning Schulzrinne. DotSlash: Handling web hotspots at dynamic content web sites. In *IEEE Global Internet Symposium*, Miami, Florida, March 2005.