High Availability and Scalability Support for Web Applications

Louis Rilling[1], Swaminathan Sivasubramanian[2], Guillaume Pierre[2]
IRISA[1] Vrije Universiteit[2]
Rennes, France Amsterdam, The Netherlands
Email: louis.rilling@irisa.fr, {swami,gpierre}@cs.vu.nl

Abstract

A database query caching technique, GlobeCBC, can be used to improve the scalability of Web applications. This paper addresses the availability issues in GlobeCBC. Even though high availability is achieved by adding more resources, proper algorithms must be designed to ensure that the clients receive consistent responses amidst failures of the edge and origin servers. We present lightweight algorithms to detect and correct server failures while providing read-your-writes consistency. They exploit the fact that the query workload of Web applications is based on a fixed set of read and write templates. We show that these algorithms incur very low overhead using several microbenchmarks and a complete Web application benchmark.

1 Introduction

Thanks to the expansion of the Internet, an ever-growing number of businesses and individuals rely on the World-Wide Web for their daily activity. As a consequence, the Web has evolved from a simple information repository into a support for complex applications. The contents to be returned to the clients are often generated dynamically each time a request is received. Dynamic Web content generation allows servers to deliver personalized contents to each user, and to take action when specific requests are issued, such as ordering an item from an e-commerce site.

Dynamic Web applications are often organized along a three-tiered architecture. When a request is issued, the Web server invokes application-specific code, which generates the content to be delivered to the client. This code, in turn, issues queries to a database where the state is preserved.

The importance that Web applications have taken in our lives raise the level of expectation that they should meet. In particular, we claim that Web application support should be both scalable enough to provide good performance under high load, and highly available even in the presence of server failures. Indeed, a highly available but unscalable application is equally useless to its users as a theoretically scalable but rarely functioning one.

Most solutions for providing scalability or availability in such context involve data caching or replication. However, different forms of replication are used for scalability and availability purposes. In particular, the virtual synchrony or transaction-based mechanisms used for fault-tolerance are considered to scale badly over large networks such as the Internet, and therefore cannot be used for Web applications.

Very few research has been conducted on providing both scalability and high availability to dynamic Web applications. A notable exception is [7], which proposes to tailor the replication to the specific data access patterns of the application. Although this approach allows to provide good scalability and availability, it requires significant expertise from application programmers.

We propose a different approach which does not require specific knowledge from the programmers. Our work is based on a database query caching system which has been shown to provide good scalability for Web applications [13]. Query caching allows edge servers to store the results of previous database queries, thereby improving the performance of future identical queries. When an update is issued on the database, invalidations are issued to the caches so that consistency is preserved.

We must address two questions. First, when an edge server fails and its clients are redirected to another edge server, we must make sure that the failover remains transparent to the clients. This requires to provide read-your-write consistency guarantees. Second, when the origin database server fails, another one must be ready to take over. While maintaining the availability of the origin database can be reduced to a classical replicated database problem, we must also make sure that no cache invalidation is lost so that application consistency is preserved. We show that these two questions can be answered with few changes in GlobeCBC, and that the resulting performance cost is minimal.

This paper is organized as follows. Section 2 presents GlobeCBC and its availability issues. Sections 3 and 4 present our solutions to maintain read-your-writes consis-

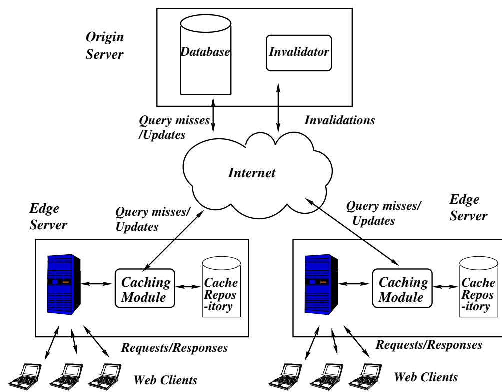

Figure 1. System Architecture.

tency during edge server and origin server failures. Finally, Section 5 presents the related work and Section 6 concludes.

2 GlobeCBC: Scalable Query Caching for Web Applications

For hosting Web applications scalably, several systems replicate their application code at multiple edge servers and keep the data centralized [1]. This allows each edge server to generate content close to the clients and spreads the computational load across multiple servers. However, simple code replication leads to the database becoming the performance bottleneck. To alleviate this bottleneck, we have built a query caching middleware, GlobeCBC, which caches the database query responses at the edge servers. It also ensures the consistency of cached responses when the underlying database is updated. We provide a brief overview here and refer the readers to [13] for more details.

2.1 Architecture

As shown in Figure 1, in GlobeCBC an application is hosted by edge servers located in different regions of the Internet. Communication between edge servers usually traverses the wide-area network thereby incurring wide-area latency. Each client is assumed to be redirected to its closest edge server using DNS-based redirection [5, 6]. We assume that the application code is replicated at all edge servers. Furthermore, one edge server acts as the *origin server*. The origin server hosts the complete database of the application.

The two key components of our system are the *caching module* and the *invalidator*. We will discuss the design and implementation of these components below.

2.2 Caching Module

The caching module is the middleware that runs in each edge server and intercepts all database queries between the application code and the origin database server. The caching module component is responsible for managing the cached query responses. When a read query is issued, it checks if the result is cached locally (using the query itself as the identifier). If found, then the result is returned immediately. Otherwise, the query is executed at the origin server. Upon successful execution, the result is stored in the local repository. Unlike the traditional database caching systems such as DBProxy [2] and DBCache [3], the caching module does not merge the query results but rather stores them independently. The caching module thus avoids the overhead of query containment checks and can return query results immediately upon a cache hit without query planning and execution overheads.

2.3 Invalidation

Caching database query responses introduces a consistency problem as a cached query response might become inconsistent when the underlying database of the service is updated. To simplify the maintenance of consistency, we assume that the query workload to the database consists of a fixed set of read and write query templates. A query template is a parametrized SQL query whose parameter values are passed to the system at runtime. This scheme is deployed, for example, using Java's prepared statement.

GlobeCBC expects the developer to specify which query template conflicts with which update template. For example, consider the template $QT1$: "SELECT price, stock from book where id=?" and its conflicting update query template $UT1$: "Update price=price+1 from books where id=?". The invalidator running in the origin server invalidates stale cached responses in the edge servers by maintaining such a conflict map for each write query template. When the invalidator receives an update query that is an instance of $UT1$, it sends out a message to invalidate all instances of $QT1$. The invalidation messages are sent asynchronously to reduce the query response time.

We have evaluated the performance of GlobeCBC compared to other data replication and query caching systems for different kinds of applications over an emulated wide-area network testbed. Figure 2 shows the results of one such experiment with the RUBBoS benchmark for one edge server and one origin server. We studied the client latency of different replication and caching strategies for different client workloads. As seen in the figure, GlobeCBC(indicated as content-blind) performs the best in terms of client latency. It can also be seen that GlobeCBC sustains higher load than other systems. We do not include more results on the performance of GlobeCBC due to space constraints. For more results with multiple edge servers and the performance of weak consistency protocols for different applications, we refer the readers to [13].

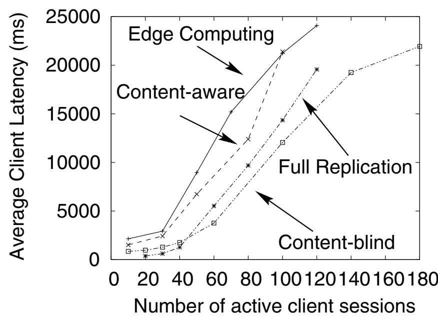

Figure 2. Effect of load on RUBBoS

2.4 Availability Issues

Two types of servers failures need to be addressed[1]. The first one is an edge server failure. When an edge server fails, its clients must be re-directed to another available server, and the failover must be transparent to the clients. While DNS-based highly available request routing is relatively well understood [5, 6], ensuring that the clients experience the same level of data consistency across all edge servers is harder. Since invalidations are sent asynchronously to the edge servers, it is possible that the results cached across different edge servers are temporarily inconsistent at any given time. In such cases, simply employing a fault-tolerant redirection system can lead to a scenario where clients are redirected to a server whose cached query responses are stale compared to those seen by them in the past. This can result in situations where the latest updates issued by the client seem to have mysteriously disappeared. To prevent such scenarios, the system must provide "read-your-writes" (RYW) consistency. This consistency model guarantees that when a client performs an update, any subsequent reads from the same client will return the effects of its update (and also possibly others) [15].

The second type of server failure is that of the origin server. As noted earlier, an origin server runs the database server which stores the application data, and the invalidator service which is in charge of issuing invalidations when the underlying database is updated. The failure of origin server's database server can be handled by classical fault-tolerant replicated database solutions. Database systems like MySQL and PgSQL have mature cluster solutions to replicate the data across multiple servers and to keep them consistent during updates. However, these solutions only address the availability of the database and do not address the availability of the invalidator. An invalidator failure can result in lost invalidations, thereby leading to stale query results being used to build client pages. Similar to databases, we must therefore replicate the invalidator service across multiple servers so that the failure of one server does not result in loss of invalidation messages.

[1]Failures due to network outages and partitions are beyond the scope of this paper.

3 Handling Edge Server Failures

3.1 Assumptions

Before presenting our solutions, we first detail our system assumptions. First, we assume the presence of a highly available DNS redirection system to resolve client requests to an available edge server [5, 6]. Second, we do not address the case when a server fails while processing a client request. In such a scenario, we assume that clients re-issue the request later and henceforth are redirected to another edge server. Third, we do not address byzantine failures, i.e., we assume a fail-stop failure model of edge servers. We do not consider the case of partial server failures that lead to returning erroneous responses.

3.2 Solution Overview

When a client is redirected to a new edge server, the edge servers need to ensure RYW consistency. A simple solution is to increment a version counter each time the database is updated, attach the latest database version number to each generated page, and ensure that the version of pages returned to a given client be higher or equal to the latest one seen by this client. However, this scheme can be very expensive as it practically requires edge servers to erase their entire cache content each time a database update occurs.

The idea behind our proposed solution is to exploit the fact that the application's query workload consists of a fixed set of read and write query templates whose conflicts are made explicit. Instead of versioning the whole database, we can maintain a *separate version number per write template*, and apply version vector-based cache consistency management. Each write query template of the application is then assigned a clock that identifies versions of the data modified by invocations of this template. For instance, in the example of Section 2.3, a clock will be assigned to the template $UT1$. If an origin server has already processed two queries of type $UT1$, then the clock will be $< UT_1, 2 >$. We then ensure that each generated page is attached with the clocks of the queries having lastly modified the data in the page. When a client issues a request to another edge server, to provide RYW the server needs to ensure that the clock values of its cached responses are at least as high as those seen by the client before. Once a violation is detected, it is sufficient for the edge server to erase the stale response from its cache, and treat the query as a cache miss.

Clocks are generated by the origin server each time it runs a write query, and are included in each query response to edge servers. This guarantees that: (i) edge servers can

keep track of the versions stored in their caches; and (ii) clients can impose minimal versions to ensure RYW consistency. This means that, instead of directly accessing the origin server database, edge servers access the invalidator service in the origin server, which forwards the queries to the database. The invalidator acts as a proxy and is responsible for generating the clocks and the invalidations.

The invalidator maintains a *template clock* TC_w for each write query template w, and updates TC_w to a strictly greater value after each execution of a write query of template w. The new value of TC_w is attached to the response. When the origin receives a read query of template r, it first collects the values of all the template clocks of write query templates that conflict with r. The origin executes the query, and includes the list of (write template id, clock value collected) pairs into the query response. When it receives the response, the edge server stores this clock list together with the query result in its cache. For instance, to a read query of type $QT1$ can be attached the clock $< UT1, \mathrm{TC}_{UT1} >$.

This algorithm ensures the following property. Consider an edge server E handling a request from client C, which results in executing a read query Q of template r, for which it has a result R stored in the cache. Let $[< w_1, c_1 >, < w_2, c_2 >, \ldots, < w_k, c_k >]$ be the clock list attached to result R, and $q_1, q_2, \ldots, q_l$ be the write queries resulting from the requests of client C so far. We note $t_1, t_2, \ldots, t_l$ the respective template ids of queries $q_1, q_2, \ldots, q_l$, and $c'_1, c'_2, \ldots, c'_l$ the respective clocks attached to the responses. If $\forall < w, t >, (w_w = t_t) \Rightarrow c_w \geq c'_t$, that is updates from client C that conflict with queries of template r have only been performed before cached result R was read in the origin database, then result R can be returned to query Q without compromising RYW consistency.

An edge server can provide RYW consistency if it knows the latest clocks of all write query templates that have been performed on behalf of this client. To maintain such information in a scalable and transparent manner, edge servers maintain a cookie at the clients' browsers in which they store the latest template clocks returned by the origin with the write query results for this client. This is scalable because neither the edge servers nor the origin have to record information about individual clients. Besides, clients only store clocks corresponding to their own requests and whose number is limited to the number of write query templates of the application. This is transparent since cookies are a standard mechanism that all popular web browsers support.

4 Handling Origin Server Failures

4.1 Assumptions

We do not treat the availability of the origin database itself. Instead, we rely on the fault-tolerance features provided by existing database replication middlewares. This makes our solution light-weight and database-agnostic. Our implementation relies on PgSQL, which ensures the consistency of replicated data across multiple servers in a local area network. We thus end up with a system model with multiple origin database and invalidator servers, and organized into a primary-backup scheme. Note that it is not necessary to run the replicated invalidators and databases on the same machines. Similarly to the edge-server failures, we assume a fail-stop failure model and do not consider the cases of requests being processed during a server failure.

4.2 Solution

To make the invalidator service fault-tolerant at a reasonable cost and with a reasonable transparency, we replicate it using a primary-backup scheme. In this scheme, one invalidator server is elected as the primary. Edge servers are made to issue all their database updates to the primary server, which is also responsible for generating the clocks for each template and propagating the invalidation messages. When the primary fails, a new primary is chosen among the existing backup invalidators.

Implementing primary-backup replication requires addressing the issues of group management and of generating version numbers consistently (especially during the transition from one primary to another). Group management mechanisms are essential as all the replicas need to agree upon the set of available servers and which one among them should act as the primary. Consistency-related mechanisms are required to ensure that version numbers are generated appropriately and that failure of an invalidator (primary or backup) does not result in lost invalidations.

4.2.1 Group Management

In our system, each invalidator server is assigned a unique numerical rank. The live (i.e., available) server having the lowest rank is always chosen as the primary. The edge servers have the list of all (alive or not) origins, and monitor them using periodic heart beat messages to identify the live replica having the lowest rank. This ensures that most connections from edge servers are addressed to the primary server. However, it may happen that a server with low rank has recovered recently, and certain requests are still issued to the former primary. In this case, the addressed origin aborts the connection, which triggers the edge server to reissue a connection to the new primary origin.

The origin servers use a group membership protocol similar to [8]. The live replicas agree on the sequence of views containing the list of available servers. The initial state of the view contains all replicas. The view is changed each time a replica suspects other replicas of having failed or recovered. A view change can only be triggered by replica

servers present in the current view. After triggering a view change, all replicas of the current view agree on the next view, and then install the new view. Subsequently, the replicas present in both views change the recorded set of view members; failed replicas leaving the view stop participating in the group management protocol; recovered replicas joining the view start participating in the protocol.

After installing a new view, the replica having the lowest rank in the view becomes primary, and the others become backup servers. Since the view is kept consistent by the group membership algorithm, the primary selection is consistent across all servers, and only one node can act as primary in a view. Note that a more sophisticated algorithm would be necessary to take network partitions into account.

4.2.2 Consistency Management

When a primary server fails and a backup takes over, we need to ensure that the new primary does not miss recent database updates (and their corresponding invalidations). Failure to do so could result in generating lower clock values than the last ones generated by the failed primary server, therefore violating RYW consistency.

This problem could easily be addressed by synchronously multicasting the updates to all the invalidators (e.g., using view synchronous multicast algorithms [9]), and running an agreement algorithm such as two-phase commit for each query of all the invalidators. However, running a synchronous multicast and agreement protocol for answering each query can severely impact the system performance as these protocols require several round-trip communications between the replica servers, thereby incurring significant overhead for both read and write queries. In our system, we take advantage of the fact that we do not require such strong consistency guarantees provided by view synchronous multicast algorithms. Instead, we adopt a simpler approach that incurs zero overhead for read queries and only one round-trip latency for write queries.

In our scheme, when a read query is received by the primary, the query is executed by one of the database replicas (as determined by the load balancer algorithms of the database replicator). Template clocks are always generated by the primary invalidator. The primary invalidator handles a write query in four steps: First it runs the query on the database and waits for the result. Second, it generates and propagates the new template clock to all invalidators, and wait for a majority of them to acknowledge receipt. We refer to this second step as *clock stabilization*. Third, it returns the query result to the requesting edge server. Fourth, it issues the necessary invalidations to the edge servers.

In the event of a primary failure, the remaining invalidators must be able to rebuild the clocks for each template to detect if any invalidation may have been lost during the failure. To this end, each replica server maintains two arrays: rec_updates and inv_done. The rec_updates array stores the latest clock values for each write template. The inv_done array stores the last clock values of each write template for which the invalidations have been sent. At a given instant if both arrays are identical, it implies that the invalidator has dispatched invalidation messages for all updates seen by the origin server till then. Similarly, for instance where the clock value for $UT1$ in inv_done is $< UT1, n >$ and its clock value in rec_updates is $< UT1, n+1 >$, then it implies that the invalidator has not yet sent out an invalidation message for invalidating read queries (cached at the edge server) conflicting with the latest write query of type $UT1$ processed by the origin server.

When the primary stabilizes a new value for a template clock, the primary and the backups update the array rec_updates to store the latest clock value. Once the primary has performed the corresponding invalidation to all the edge servers connected, the new clock value is then stored in the inv_done array of the primary. The primary periodically broadcasts its inv_done array to the backups. Note that it is not necessary to maintain strong consistency between views of the inv_done array.

4.2.3 Failure Handling

When a backup invalidator becomes the new primary, it executes a preliminary step before accepting to serve queries. This preliminary step includes an anti-entropy algorithm [15]. First the new primary retrieves the rec_updates and inv_done arrays of at least a majority of replicas (including itself). These replicas commit themselves to not accept any stabilization request from previous views. Then, for each write query template, the primary computes the maximum value for each array. Missing invalidations are detected by comparing the values in the two arrays: if for a write query template w rec_updates[w] is greater than inv_done[w], an invalidation of results conflicting with template w may have been lost, and the new primary reissues invalidations requests for template w at all edge servers connected. After this step, the new primary becomes active and is ready to serve new queries.

Rebuilding the clock arrays and refusing stabilization requests from previous views ensures that clock values for each template are generated in a strictly increasing order even in the presence of primary failures. However, view changes do not necessarily happen immediately at the same time for each replica (recall that we do not use a synchronous protocol for view maintenance). It is therefore possible that the former primary thinks it is still the primary, while new write queries and new values for these clocks are generated by a new primary. If the former primary accepts to serve a write query during this transition time, the query

will be blocked because stabilization at a majority of replicas is not possible after the new primary becomes active. To prevent from losing all requests received by the former primary, the former primary then forwards pending updates to the new primary when it installs the new view. The new primary recomputes template clocks for these updates, stabilizes them, and the replicas send back acknowledgments to the former primary with the new template clock. This avoids breaking the connection between an edge server and the origin server because of a view change.

5 Performance Evaluations

This section evaluates the overhead of our availability algorithms. First, we demonstrate that during edge server failures simple database caching techniques can violate RYW consistency. Subsequently, we demonstrate the low overhead of our edge server failure handling techniques. Second, we demonstrate the efficiency of the techniques to handle the failure of origin server invalidators, and show that the overhead introduced by the invalidator replication protocol is negligible.

We implemented a prototype of GlobeCBC, including its fault-tolerance and RYW consistency management algorithms. The caching module in the edge server is implemented by modifying the Apache PHP driver for PostgreSQL. The invalidator service is written as a multi-thread stand-alone server using C. We evaluated the efficiency of our system using RUBBoS, a benchmark application that models `slashdot.org`. RUBBoS application's database consists of five tables, storing information regarding users, stories, comments, submissions and moderator activities. We filled the database with information of $500,000$ users and $200,000$ comments. For our experiments, we choose the open source PHP implementation of these benchmarks[2]. The client workload for both benchmarks is generated by Emulated Browsers (EBs) and conforms to the TPC specification for clients [14]. The user workload for RUBBoS contains more than 15% interactions that lead to updates.

5.1 Occurrence of RYW Violations

We first demonstrate that simple asynchronous invalidations used in GlobeCBC can lead to RYW violations, independent of edge server failures. To this end, we plotted the progression of the data versions obtained by a single RUBBoS client only for a single template clock. In Figure 3, the continuous line represents the minimal version needed to respect RYW consistency (that is all results must have clocks above the line), and the slanting crosses represent the versions of stale cached results that would have been returned by the edge server without our clock handling algorithm (these results would have been below the line). We can see that there were 3 stale cached results detected and which required fetching new query results from the origin server. This is due to the fact that invalidations are sent to the edge server asynchronously. We also observed that there were 22 queries of the same templates for which up-to-date results were found in the cache. So, without our clock handling mechanisms a significant fraction of the responses would have violated the RYW consistency.

[2]`http://jmob.objectweb.org/rubbos.html`

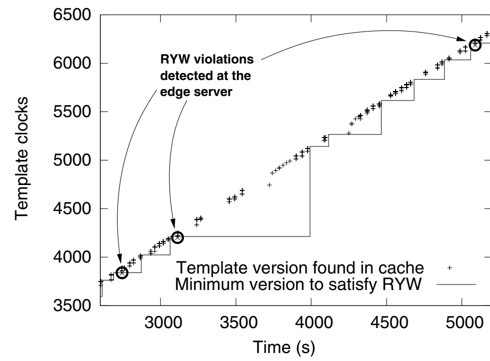

Figure 3. Version numbers of query results received by one client for one template clock.

5.2 Overhead in Edge Server Failures

To study the effect of edge server failover on clients, we ran a synthetic experiment with the RUBBoS benchmark. We made 100 clients access the application through 2 edge servers. The edge servers access the database at one origin server, which was not replicated in the experiment. To emulate a WAN configuration, we used the NIST Net network emulator to insert 0 ms latencies between the RUBBoS clients and the edge servers, and 100 ms latencies between the edge servers and the origin server. All the nodes were bi-processor 1 GHz Pentium III.

We started the experiment with the two edge servers E_1 and E_2, each being accessed by a pool of 50 clients. After 1800 seconds, we made edge server E_1 fail and redirected E_1's clients to E_2. Figures 4 and 5 show the request latencies perceived by the two pools of clients across time. We marked the periods before, during and after the failure.

Interestingly, the clients accessing E_2 from the beginning do not perceive any change in performance. This is due to the fact that one edge server is powerful enough to serve the two client populations without any drop in performance. A similar observation can be made for E_1's client population when comparing the requests well before and well after the failure. However, the queries that were already started on E_1 but not yet completed at the time of the failure show

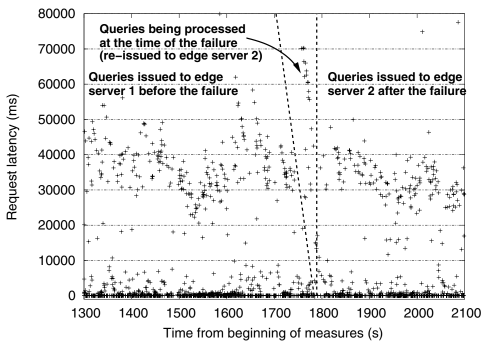

Figure 4. Request latencies of E_1's clients.

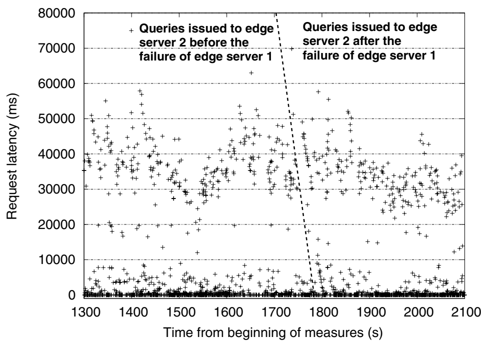

Figure 5. Request latencies of E_2's clients

a somewhat greater latency. This is due to the fact that, upon detection of the failure of E_1, clients had to re-issue their connections to E_2. We conclude that, provided that the surviving edge servers have sufficient capacity to serve the whole client population, no significant delay is added by the RYW consistency enforcement at the time of a failure.

5.3 Scalability of Invalidator Replication

To evaluate the overhead of invalidator replication, we first ran microbenchmarks to measure the overhead of executing a write and read query. In the first microbenchmark, we generated a series of write queries from three edge servers concurrently. Each edge server performs a sequence of write queries (insert a record in a small database table) at maximal rate, that is a new write query is performed as soon as the result of the previous one is received. To separate the impact of query execution on our results, we do not execute the queries on the database. In this experiment, the edge servers and the invalidators are located on a same switched LAN (with round trip time lower than 1 ms), which makes edge servers issue queries at high rate. Each server runs a bi-processor 1 GHz Pentium III.

The cumulative distributions of the times to serve write queries at the primary invalidator are shown on figure 6. All distributions are closely gathered around their median. With

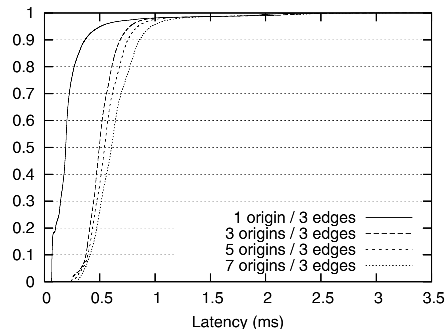

Figure 6. Impact of the number of invalidators on the latency of write queries.

only one invalidator replica, the median is $192\mu s$, whereas the medians are $496\mu s$, $544\mu s$, and $607\mu s$ for 3, 5, and 7 invalidator replicas respectively. The difference in latency between one and more than one replicas (albeit only small) is due to the LAN round trip latency incurred in the clock stabilization protocol. In all cases, the latencies are very low compared to WAN latencies (which is in the order of 10s of milliseconds) and suggest that invalidator replication can support heavy client workloads. Moreover, we observe that the increase in number of replicas (of the invalidator) does not affect the performance.

The second microbenchmark consists of only read queries. The distributions obtained are not shown due to space constraints, but we observe that the median latency is in the order of $200\mu s$. In other words, we observe that the replication of invalidator does not induce any overhead on read queries as read queries are never blocked in our replication protocol.

To confirm that invalidator replication induces only negligible overhead for a real application, we ran the RUBBoS benchmark for 20 and 100 active clients with a single edge server and 1 to 5 invalidator replicas. The database and invalidators are run on different servers and the configuration of servers are the same as the one used in our previous experiments. The cumulative distribution of client request latencies measured for 20 clients is shown on figure 7. From this results, it can be seen that the distributions are very close to each other and this shows that invalidator replication has only a negligible overhead. The results with 100 clients are similar, except that the latencies are higher because of the increase in query load.

6 Related Work

In replicated systems, there is a constant tradeoff between scalability and availability. Various systems have been built to host Web applications [1, 2, 3, 4, 7, 10, 11, 12,

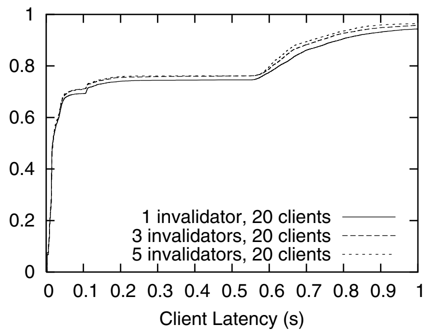

Figure 7. The impact of invalidator replication on RUBBoS performance.

13]. However, most of these systems focus on scalability only and do not address availability issues (or vice versa). For example, database caching systems such as [2] only aim at improving the scalability of Web applications and do not address the problem of edge server or origin server failures. As described earlier, replicating the origin database for fault-tolerance is not sufficient to guarantee RYW consistency amidst failures. On the other extreme, systems such as [1] employ highly-available database replication techniques at the origin server only and provide excellent consistency guarantees. However, such solutions come at a cost of reduced performance as each database query from the application needs to traverse the wide-area network to the origin server thereby incurring a WAN latency.

To our knowledge, only one system focuses on both scalability and availability of Web applications [7]. The authors propose to tailor the replication strategies to the data access patterns of the hosted application. This allows to exploit detailed knowledge of the application semantics and choose the right balance between availability and scalability. However, even though this approach can provide good scalability and availability, it requires significant expertise from the application developers to understand the impact of different data replication and fault tolerance protocols, and to choose the optimal tradeoff point. In contrast, our system does not require any application-specific knowledge from the developer other than defining query template conflicts. The availability techniques themselves are totally transparent to the developer.

7 Conclusions

This article demonstrates that Web application hosting can be both scalable and highly available. Many database caching systems exploit the fact that Web applications are composed of a fixed set of query templates. We used this property to devise lightweight algorithms that provide read-your-writes consistency even across edge server and origin server failures. We showed that RYW violations do occur relatively frequently, and that the fault-tolerant algorithms to detect and correct them imposes a very low overhead compared to the cost of a database query execution.

Our work relies on off-the-shelf fault-tolerant database replication features. However, we believe that the performance of the generic replication algorithms used in these systems can be improved by once again exploiting knowledge about query templates used by the applications. We are therefore currently working on Web application specific database replication protocols that will allow even more scalability without compromising high availability.

References

[1] Akamai Inc. Edge Computing Infrastructure.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In *Proc. ICDE*, Mar. 2003.

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

[4] E. Cecchet. C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26, June 2004.

[5] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[6] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proc. INFOCOM*, Mar. 1998.

[7] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proc. Intl. WWW conf.*, May 2003.

[8] F. Greve, M. Hurfin, M. Raynal, and F. Tronel. Primary component asynchronous group membership as an instance of a generic agreement framework. Technical Report RR-3856, INRIA, Rennes, France, 2000.

[9] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, Apr. 1997.

[10] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proc. Middleware*, Toronto, Canada, Oct. 2004.

[11] M. Rabinovich, Z. Xiao, and A. Agarwal. Computing on the edge: A platform for replicating internet applications. In *Proc. WCW Workshop*, Hawthorne, NY, USA, Sept. 2003.

[12] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: autonomic data replication for web applications. In *Proc. Intl. WWW Conf.*, New York, NY, USA, 2005.

[13] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands, June 2006.

[14] W. Smith. TPC-W: Benchmarking an e-commerce solution. http://www.tpc.org/tpcw/tpcw_ex.asp.

[15] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welsh. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. Intl. PDIS Conf.*, Austin, TX, Sept. 1994.