GlobeDB: Autonomic Data Replication for Web Applications

Swaminathan Sivasubramanian[1] Gustavo Alonso[2] Guillaume Pierre[1] Maarten van Steen[1]

Dept. of Computer Science[1]
Vrije Universiteit
Amsterdam, The Netherlands
{swami,gpierre,steen}@cs.vu.nl

Dept. of Computer Science[2]
Swiss Federal Institute of Technology (ETHZ)
Zurich, Switzerland
alonso@inf.ethz.ch

ABSTRACT

We present GlobeDB, a system for hosting Web applications that performs autonomic replication of application data. GlobeDB offers data-intensive Web applications the benefits of low access latencies and reduced update traffic. The major distinction in our system compared to existing edge computing infrastructures is that the process of distribution and replication of application data is handled by the system automatically with very little manual administration. We show that significant performance gains can be obtained this way. Performance evaluations with the TPC-W benchmark over an emulated wide-area network show that GlobeDB reduces latencies by a factor of 4 compared to non-replicated systems and reduces update traffic by a factor of 6 compared to fully replicated systems.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]: Design studies; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Design, Measurement, Performance

Keywords

Data Replication, Edge Services, Autonomic Replication, Performance

1. INTRODUCTION

Edge service architectures have become the most widespread platform for distributing Web content over the Internet. Commercial Content Delivery Networks (CDNs) like Akamai [4] and Speedera [28] deploy edge servers around the Internet that locally cache Web pages and deliver them to the clients. By serving Web content from edge servers located close to the clients, the response time for serving clients is reduced as each request need not travel across a wide-area network. Typically, edge servers store static Web pages and a plethora of techniques and commercial systems exist for replicating static pages [26].

In practice, however, a large amount of Web content is generated dynamically. These pages are generated upon request using Web applications that take, e.g., individual user profiles and request parameters, into account when producing the content. These applications tend to run on top of databases. When a request arrives, the application examines the requests, issues the necessary read or update transactions to the database, retrieves the data, and composes the page that is sent back to the client. To speed up the access, traditional CDNs use techniques such as fragment caching whereby the static fragments (and sometimes also certain dynamic parts) of a page are cached at the edge servers [13, 23, 9, 18, 11]. However, this solution assumes that temporal locality of requests is high and database updates are infrequent. Applications that do not meet these assumptions require different solutions than fragment caching.

In this paper we explore a different approach to delivering dynamic content that offers more flexibility than fragment caching. The idea is to replicate not the pages but the data residing in the underlying database. This way, the dynamic pages can be generated by the edge servers without having to forward the request to a centralized server to guarantee consistency. In particular, if the temporal locality and update rate are high, an application may benefit by moving (part of) the underlying database, along with the code that accesses the data, to the location where most updates are initiated. If the temporal locality and the update rate is low, then replication of the database and access code may be a viable solution. This allows the edge servers to generate documents locally. In both cases, the amount of wide-area traffic can be reduced, leading to better client-perceived performance.

We are thus confronted with the problem of sustaining performance through distribution and replication of (parts of) an application, depending on access and usage patterns. In general, we need to address the following three issues: (1) determine which parts of an application and its data need to be replicated, (2) find where these parts need to be placed in a wide-area network, and (3) decide how replicated data should be kept consistent in the presence of updates.

These issues are currently handled by human experts who manually partition the databases of a Web application, and subsequently distribute the data and code among various servers around the Internet. Not only is this a difficult and time-consuming process, it is also not very feasible when usage and access patterns change over time [24]. In addition, consistency in these systems tends to be ad-hoc and largely application-dependent.

In this paper we propose GlobeDB, a system for hosting Web applications, which handles distribution and partial replication of application data *automatically* and *efficiently*. Our system provides Web-based data-intensive applications the same advantages that content delivery networks offer to traditional Web sites: low latency and reduced network usage. We substantiate these claims

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

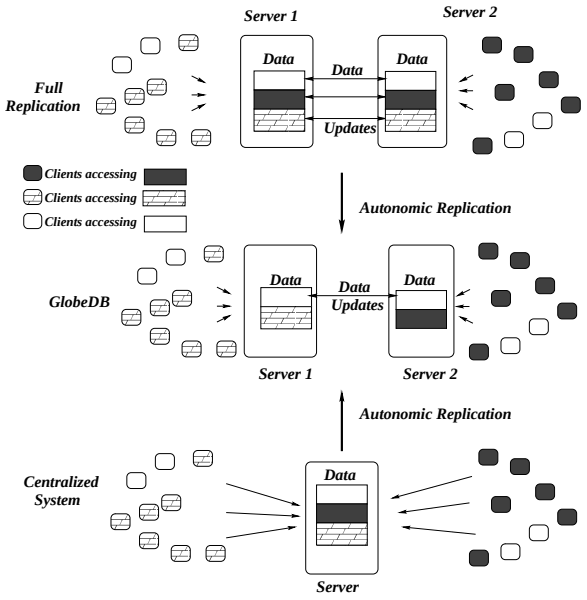


Figure 1: Example of benefits of autonomic replication.

through extensive experimentation of a prototype implementation running the TPC-W benchmark over an emulated wide-area network.

Our major contribution is that we demonstrate that configuration of Web applications for data and code replication can be largely automated, and in such a way that it yields a substantial performance improvement in comparison to simple or non-replicated approaches. As such, our work improves upon other research demonstrating that application-specific replication improves performance as well (e.g., [15]). In particular, we argue that there is often no compelling reason to adapt the logical design of an application in order to support replication. By automatically partitioning and replicating an application's associated database, we can achieve the same results without involving the application designer or requiring other human intervention.

The rest of the paper is organized as follows. Section 2 presents several design issues involved in building the system and motivates our design choices. Section 3 presents GlobeDB's architecture and Section 4 describes the design of the data driver, the central component of GlobeDB. Section 5 describes the replication and clustering algorithms adopted in our system. Section 6 presents an overview of GlobeDB implementation and its internal performance. Section 7 presents the relative performance of GlobeDB and different edge service architectures for the TPC-W benchmark. Finally, Section 8 discusses the related work and Section 9 concludes the paper.

2. DESIGN ISSUES

To illustrate the benefits of autonomic replication, consider the scenarios presented in Figure 1 that show edge server(s) hosting a Web application. As seen in the figure, there is a fraction of data accessed only by clients of server 1, another fraction by clients of server 2 and the rest are accessed by clients of both servers. Not replicating at all (centralized system) can result in poor client response time. Replicating all data everywhere (full replication) can result in significant update traffic between servers for data they barely access. In such scenarios, GlobeDB can be very useful as it places the data in only those servers that access them often. This can result in a significant reduction in update traffic and improved client-perceived response time.

Building a system for autonomic replication of Web application data requires addressing many issues such as identifying the granularity and constituents of the data segments, finding the optimal placements for each data unit, and maintaining consistency of replicated data units. In this section, we discuss these issues in detail.

2.1 Application Transparency

The first and foremost issue is to decide the extent to which an application should be aware of data replication. Replication can yield the best performance if it is completely tuned to the specific application and its access patterns [15]. However, this requires significant effort and expertise from an application developer. As a consequence, optimal performance is often not reached in practice. Furthermore, changes in access pattern may warrant changes in in the replication strategies, thereby making the design of an optimal strategy even more complex.

In our system, we chose a *transparent* replication model. The application developer need not worry about replication issues and can just stick to the functional aspects of the application. The system will automatically find a placement and replication strategy, and adapt it to changing access patterns when needed.

2.2 Granularity of Data

The underlying principle behind our system is to place each data unit only where it is accessed. In our previous research on replication for static Web pages, we showed that the optimal replication performance in terms of both client-perceived latency and update bandwidth can be achieved if each Web page is replicated according to its individual access patterns [20]. A naive transposition of this result would lead to replicating each database record individually. However, such fine-grained replication can result in significant overhead as the system must maintain replication information for each record.

In our system, we employ an approach where the data units are initially defined at a very fine grain. Data units having similar access patterns are then automatically clustered by the system. The system subsequently handles replication at the cluster level, thereby making the problem of managing a cluster feasible without losing the benefits of partial replication. However, a caveat of this approach is that if the access patterns change, then the system must perform re-clustering to sustain good performance. More information on the clustering algorithm used in our system is presented in Section 5.

2.3 Consistency

One important issue in any replicated system is consistency. Consistency management has two main aspects: update propagation and concurrency control. In update propagation, the issue is to decide which strategy must be used to propagate updates to replicas. Many strategies have been proposed to address this issue. They can be widely classified into push-based and pull-based strategies. Pull-based strategies are mostly suitable for avoiding unnecessary data update transfers when no data access occurs between two subsequent updates. In our system, we decided to use a *push* strategy where all updates to a data unit are pushed to the replicas immediately. Pushing data updates immediately ensures that replicas are kept consistent and that the servers hosting replicas can serve read requests immediately.

Yet, propagating updates is not sufficient to maintain data consistency. The system must also handle concurrent updates to a data unit emerging from multiple servers. Traditional non-replicated DBMSs perform concurrency control using locks or multiversion models [16]. For this, a database requires an explicit definition of *transaction*, which contains a sequence of read/write operations to

a database. For example, PostgreSQL uses a variant of multiversion model called snapshot isolation to handle concurrent transactions [3]. When querying a database each transaction sees a snapshot of consistent data (a database version), regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data produced by concurrently running update transactions.

However, concurrency control at the database level can serialize updates only to a single database and does not handle concurrent updates to replicated data units at multiple edge servers. Traditional solutions such as two-phase commit are rather expensive as they require global locking of all databases, thereby reducing the performance gains of replication. To handle this scenario, the system must serialize concurrent updates from multiple locations to a single location. GlobeDB does not guarantee full transaction semantics, but enforces consistency for each query individually. In other words, it imposes that each transaction be composed of a single query/operation which at most modifies a single data unit. The concurrency control adopted in our system is explained in Section 3.

2.4 Data Placement

Automatic data placement requires the system to find a set of edge servers to host the replicas of a data cluster according to certain performance criteria. One can measure the system performance by metrics such as average read latency, average write latency, amount of update traffic, etc. But a naive approach based on optimizing the system performance for one of these metrics *alone* can easily result in degrading the performance according to other metrics. For example, a system can be optimized for minimizing read latency by replicating the data to all replica servers. However, this can lead to huge update traffic if the number of updates is high.

In general, there is a clear tradeoff between the performance gain due to replication and the performance loss due to consistency enforcement. However, there is no universal definition of "best" tradeoff. In fact, each system administrator should specify a particular tradeoff based on the system needs. For example, the administrator of a CDN with (theoretically) unlimited bandwidth may choose to optimize on client response time alone. The same administrator, when facing a bottleneck at the central server, may prefer to minimize update traffic.

In our system, the system administrator specifies relative performance tradeoffs as the weights of a *cost function*. This function aggregates multiple performance metrics into a single abstract metric. Optimizing the cost function is equivalent to optimizing the global system performance. This function therefore acts as a measure of the desired system performance and aids the system in making its placement decisions. The cost function and placement algorithms are presented in Section 5.

3. SYSTEM ARCHITECTURE

3.1 Application Model

The application model of our system is shown in Figure 2. An application is made of code and data. The code is written using standard dynamic Web page technology such as PHP and is hosted by the Web server (or the Web application server). It is executed each time the Web server receives an HTTP request from its clients, and issues read/write accesses to the relevant data in the database to generate a response.

Access to the data is done through a data driver which acts as the interface between code and data.[1] The data driver preserves

[1]This driver is different from conventional PHP drivers as it not just an interface driver but also responsible for functional aspects such as location of replicated data.

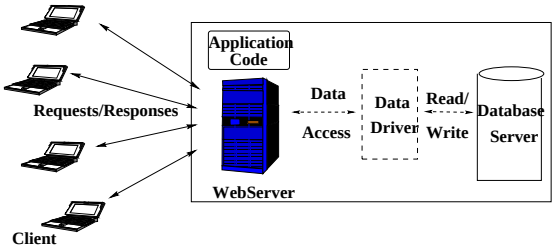


Figure 2: Application Model

distribution transparency of the data and has the same PHP interface as regular PHP drivers. It is responsible for finding the data required by the code, either locally or from a remote server, and for maintaining data consistency.

We assume that the database is split into n data units, D_1, D_2,$\cdots$, D_n, where a data unit is the smallest granule of replication. Each unit is assumed to have a unique identifier, which is used by the data driver to track it. An example of a data unit is a database record identified by its primary key.

GlobeDB enforces consistency among replicated data units using a simple master-slave protocol: each data cluster has one master server responsible for serializing concurrent updates emerging from different replicas. GlobeDB assumes that each database transaction is composed of a single query which modifies at most a single data unit. When a server receives an update request, it forwards the request to the master of the cluster, which processes the update request and propagates the result to the replicas. Note that this model is sometimes called eventual consistency. If stronger consistency is needed, then models such as session guarantees can be envisaged [29].

3.2 System Architecture

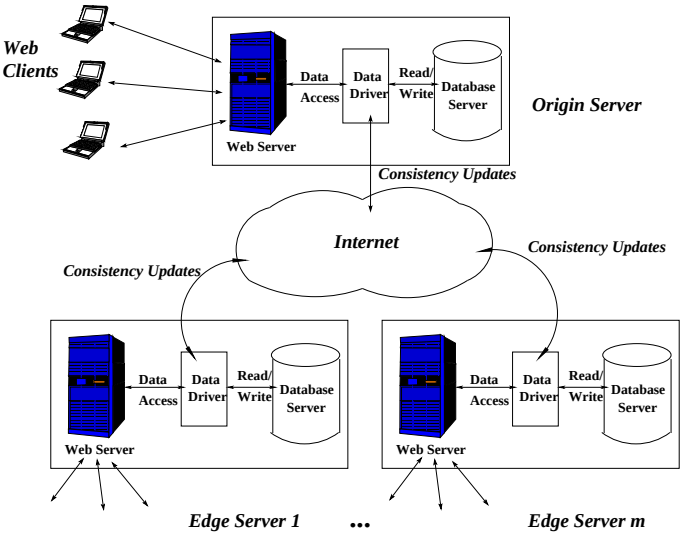


Figure 3: System Architecture - Edge servers serving clients close to them and interactions among edge servers goes through Wide-area network.

The architecture of GlobeDB is presented in Figure 3. An application is hosted by m edge servers spread across the Internet. Communication between edge servers is through wide-area networks

incurring wide-area latency. Each client is assumed to be redirected to its closest edge server using enhanced DNS-based redirection [12, 14]. Furthermore, for each session, a client is assumed to be served by only one Web server.

When a client issues an HTTP request to a Web server, the request is forwarded to the application code (e.g., PHP) residing in the server. The application code usually issues a number of read or write accesses to its data through the data driver. The application data are partially replicated, so the local database hosts only a subset of all data clusters. The data driver is responsible for finding the relevant data either locally or from a remote edge server if the data are not present locally. Additionally, when handling write data accesses, the driver is also responsible for ensuring consistency with other replicas.

To perform autonomic replication, the system must decide on the placement of replicas for each data cluster and choose its master according to its access pattern. To this end, each application is assigned one *origin server*, which is responsible for making all application-wide decisions such as clustering data units and placing clusters on edge servers. The origin server performs replica placement periodically to handle changes in data access patterns and the creation of new data units. For reasons explained below, the origin server also has a full replica of the database.

Consistency is enforced using a simple master-slave protocol: each data cluster has one master server responsible for serializing concurrent updates emerging from different replicas. When a server holding a replica of a data cluster receives a read request, it is answered locally. When it receives an update request, it is forwarded to the master of the data cluster. If the server does not have a replica, then requests are forwarded to the origin server. More information on locating data units is presented in Section 4.

Note that eventual consistency does not guarantee that all replicas are identical at all times. The master of a cluster delivers updates to all replicas in the same order without any global locking. This can lead to transient situations where the latest updates have been applied only to a fraction of the replicas. We assume that this can be tolerated by the application as each client session is handled by only one edge server in the network. Note that even existing solutions over commercial database systems such as DBCache [6] and MTCache [17] face similar issues and guarantee the same level of consistency as GlobeDB.

4. DATA DRIVER

The data driver is the central component of GlobeDB. It is in charge of locating the data units required by the application code and maintaining consistency of the replicated data.

4.1 Types of Queries

The primary functionality of the data driver is to locate the data units required by the application code. Data access queries can be classified into write and read queries based on whether or not an update to the underlying database takes place. Another way of classifying queries is based on the number of rows matched by the query selection criterion. We refer to the queries based on primary keys of a table or that result in an exact match for only one record as *simple queries*. An example of a simple query is "Find the customer record whose userid is 'xyz'." Queries based on secondary keys and queries spanning multiple tables are referred to as *complex queries*. An example of a complex query is "Find all customer records whose location is 'Amsterdam'."

As noted earlier, we assume that each data unit has a unique identifier. For fine-grained data units, such as database records, we use the primary key as the record's unique identifier. This allows the data driver to map simple queries to required data units, which makes locating data units relatively straightforward.

For answering a complex query, the driver cannot restrict its search to its local database but needs to inspect the entire database. Such a query can be answered by forwarding it to a subset of servers that jointly have the complete database table. For the sake of simplicity, we stipulate that the origin server has a replica of all data units. For this reason all complex queries are forwarded to the origin server which will incur a wide-area latency.

At the outset, being able to answer only simple queries locally might appear very limiting. However, typical Web applications issue a majority of simple queries. To get an idea of the percentage of simple queries used on real e-commerce applications, we examined the TPC-W benchmark which models a digital bookstore [27]. We collected the SQL traces generated by execution of the benchmark's ordering mix workload [1]. We analyzed the accesses to the book tables (which contains the records pertaining information about individual books) and found that more than 80% of the accesses to the table consist of simple queries or can be easily rewritten as simple queries. Similarly, about 80% of accesses to the customer tables use simple queries. Similar figures are seen for other workload mixes of TPC-W. In TPC-W, updates to a database are always made using simple query.

4.2 Locating Data Units

```
if complex query then
    Execute query at the origin server and return result;
else
    if read then
        Execute query locally;
        if execution returns result then
            return result;
        else
            execute on origin server and return result;
        end
    else
        Get cluster id of data unit from (local or origin server);
        Find master for cluster from cluster-property table;
        Execute query on master server and return result;
    end
end
```

Algorithm 1: Pseudocode used by data driver for executing queries

The data driver of each edge server maintains three tables. The *cluster-membership* table stores the identifiers of data units contained in each cluster. The *cluster-property* table contains the following information for each data cluster: the origin server, the master replica, and the list of servers that host a copy of this cluster. These two tables are fully replicated at all edge servers. Each driver also maintains an *access table* to keep track of the number of read and write accesses to each cluster.

To answer simple queries, the driver locates a data unit by identifying the cluster to which the data unit belongs, using the *cluster-membership* table. Once the appropriate cluster is identified, the driver uses the *cluster-property* table to find details about the location of the cluster and its master. Upon each read or write access, the driver updates the access table accordingly. The data driver forwards all complex queries to the origin server.

A naive design of the *cluster-membership* table can be a scalability bottleneck. In our initial implementation, we used bit arrays for numerical primary key IDs and bloom filters for non-numerical IDs [5]. A typical bit-array based cluster membership table will have a size of only 125 Kbits for each cluster to represent a database

with a million data units. Such a small size allows the table to reside in main memory thereby resulting in faster access. However, these filters have several disadvantages. First, storing non-numerical IDs using Bloom filters can result in potential inaccuracies that result in redundant network traffic. Second, the system needs to allocate enough memory for filters to accommodate creation of new data units in the future, which poses a scalability problem.

To overcome these shortcomings, in our current implementation the *cluster-membership* table is stored along with the respective database records. In this implementation, each database record has an extra attribute that indicates which cluster the record belongs to. The pseudocode for executing queries by the data driver is shown in Algorithm 1. As seen in the figure, simple queries with read accesses are always first executed locally. If a record is returned, the result is returned immediately.[2] Otherwise, the query is forwarded to the origin server (which contains a full copy of the database). For update queries, the driver first runs a query to find the cluster ID. If the data record is replicated locally, its cluster ID will be returned and the information regarding who its master is can be obtained from the *cluster-property* table. Then, the update query is forwarded to the master. In case of update queries to a data unit not present locally, the query to find cluster information for the data unit will return no result and this query is then run on the origin server. Subsequently, the update query will be sent to the appropriate master server. The *cluster-property* table is simply implemented as a file.

5. REPLICATION ALGORITHMS

Replicating an application requires that we replicate its code and data. For the sake of simplicity, in this paper we assume that the code is fully replicated at all replica servers. In this section, we present the algorithms we use for clustering data units, placing the data cluster, and selecting their master. For the placement of data, we use a *cost function* that allows the system administrator to tell GlobeDB his/her idea of optimal performance. As we explain in detail in this section, GlobeDB uses this function to assess the goodness of its placement decisions.

5.1 Clustering

As we mentioned earlier, in GlobeDB, data units with similar access patterns are clustered together. A similar problem was addressed in [10] in the context of clustering static Web pages to reduce the overhead in handling replicas for each Web page. The authors propose several spatial clustering algorithms to group pages into clusters and incremental clustering algorithms to handle the creation of new pages. They show that these clustering algorithms perform well for real-world Web traces. We use similar algorithms for clustering data units.

The origin server is responsible for clustering the data units during the initial stages of system. The origin server collects access patterns of data units from all edge servers into its access table. Each data unit D_i's access pattern is modelled as a $2{*}m$-dimensional vector, $A_i =< r_{i,1},r_{i,2},\cdots,r_{i,m},w_{i,1},\cdots,w_{i,m} >$, where $r_{i,j}$ and $w_{i,j}$ are respectively the number of read and write accesses made by the edge server R_j to the data unit D_i. The origin server runs a spatial clustering algorithm that uses correlation-based similarity on the access vectors of the data units. As a result, two data units D_i and D_j are grouped into the same cluster if and only if A_i and A_j are similar. In correlation-based similarity, the similarity between two data units D_i and D_j is given by:

$$Sim(i,j) = \frac{\sum_{k=1}^{2*m}(a_{i,k}-\bar{a_i})(a_{j,k}-\bar{a_j})}{\sqrt{\sum_{k=1}^{2*m}(a_{i,k}-\bar{a_i})^2 \sum_{k=1}^{2*m}(a_{j,k}-\bar{a_j})^2}}$$

where $< a_{i,1},a_{i,2},\cdots,a_{i,m} > = < r_{i,1},r_{i,2},\cdots,r_{i,m} >$ and $< a_{i,m+1},a_{i,m+2},\cdots,a_{i,2*m} > = < w_{i,1},\cdots,w_{i,m} >$. Data units D_i and D_j are clustered together if $Sim(i,j) \geq x$, for some threshold value x, where $0 \leq x \leq 1$.

The origin server iterates through data units that are yet to be clustered. If a data unit D_i is sufficiently close to the access vector of an existing cluster, it is merged into it. Otherwise, a new cluster with D_i as the only member is created. Once the data clusters are built, the origin server creates the appropriate cluster-membership table for each cluster. Obviously, the value of x has an impact on the effectiveness of the replication strategy. In all experiments from Section 6 we fix the threshold x to 95%. We will study the process of determining the optimal threshold value in the near future.

Data clustering works well if data units once clustered do not change their access pattern radically. However, if they do, then the clusters must be re-evaluated. The process of re-clustering requires mechanisms for identifying stale data units within a cluster and then re-clustering them. For reasons of space, we do not address the re-clustering problem in this paper. It is orthogonal to the replication strategy and it mainly involves determining when to re-cluster and how to re-cluster efficiently. Also note that re-clustering can be done by progressively invalidating and validating copies at the different edge servers as it is done for data caches.

5.2 Selecting a Replication Strategy

The origin server must periodically select the best replication strategy for each cluster. A replication strategy involves three aspects: *replica placement*, *consistency mechanism*, and, in our case, *master selection*. As we use push strategy as our consistency mechanism, the selection of a replication strategy for a cluster boils down to deciding about replica placement and selecting the master.

As noted earlier, to select the best replication strategy, the system administrator must specify what "best" actually means. In GlobeDB, we represent overall system performance into a single abstract figure using a *cost function*. A cost function aggregates several evaluation metrics into a single figure. By definition, the best configuration is the one with the least cost. An example of a cost function which measures the performance of a replication strategy s during a time period t is:

$cost(s,t) = \alpha * r(s,t) + \beta * w(s,t) + \gamma * b(s,t)$

where r is the average read latency, w is the average write latency, and b is the amount of bandwidth used for consistency enforcement.

The values α, β and γ are weights associated to metrics r, w, and b respectively. These weights must be set by the system administrator based on system constraints and application requirements. A larger weight implies that its associated metric has more influence in selecting the "best" strategy. For example, if the administrator wants to optimize on client performance and is not concerned with the bandwidth consumption, then weights α and β can be increased. Finding the "best" system configuration now boils down to evaluating the value of the cost function for every candidate strategy and selecting the configuration with the least cost.

In GlobeDB we use the cost function as a *decision making tool* to decide on the best server placements and master server for each cluster. Ideally, the system should periodically evaluate all possible combinations of replica placement and master server configurations for each cluster with the cluster's access pattern for the past access

[2]This is also done for exact match queries (queries that match a single data unit) based on non-primary queries.

period. The configuration with the least cost should be selected as the best strategy for the near future. This relies on the assumption that the past access patterns are a good indicator for the near future. This assumption has been shown to be true for static Web pages and we expect the dynamic content will exhibit similar behavior [20].

Ideally, the system should treat the master selection and replica placement as a single problem and select the combination of master-slave and replica placement configuration that yields the minimum cost. However, such a solution would require an exhaustive evaluation of $2^m * m$ configurations for each data cluster, if m is the number of replica servers. This makes this solution computationally infeasible. In GlobeDB, we use heuristics to perform replica placement and master selection (discussed in the next subsections). We propose a number of possible heuristics for placement and a method for optimal selection of master server. This reduces the problem of choosing a replication strategy to evaluating which combination of master server and placement heuristics performs the best in any given situation. After selecting the best replica placement and master for each data cluster, the origin server builds the cluster-property table and installs it in all edge servers.

5.3 Replica Placement Heuristics

Proper placement of data clusters is important to obtain good client latencies and reduced updated traffic. We define a family of placement heuristics Px where an edge server hosts a replica of a data cluster if its server generates at least $x\%$ of data access requests.

Obviously, the value of x affects the performance of the system. A high value of x will lead to creating no replica at all besides the origin server. On the other hand, a low value of x may lead to a fully replicated configuration.

Expecting the system administrator to determine the appropriate value for x is not reasonable, as the number of parameters that affect system performance is high. Instead, in GlobeDB, administrators are just expected to define their preferred performance tradeoffs by choosing the weight parameters of the cost function. The origin server will then evaluate the cost value for placement configurations obtained for different values of x (where x=5,10,15), and select the one that yields the least cost as the best placement configuration.

5.4 Master Selection

Master selection is essential to optimize the write latency and the amount of bandwidth utilized to maintain consistency among replicas. For example, if there is only one server that updates a data cluster, then that server should be selected as the cluster's master. This will result in low write latency and less update traffic as all updates to a cluster are sent to its master and then propagated to the replicas.

We use a method for optimal selection of master server that results in the least average write-latency. Let $w_{i,j}$ be the number of write access requests received by edge server R_j for cluster i and l_{jk} be the latency between edge server j and k (we assume that latency measurements between servers are symmetric, i.e., l_{kj}=l_{jk}). The average write latency for data cluster i whose master is k is given by: $wl_{ki} = (\sum_{j=1}^{m} w_{i,j} * l_{jk})/(\sum_{j=1}^{m} w_{i,j})$. The origin server selects the server with lowest average write latency as the master for a data cluster .

6. IMPLEMENTATION AND ITS PERFORMANCE

In this section, we discuss the salient components of our prototype system and also present performance measurements on the overhead that the data driver introduces to a single edge server.

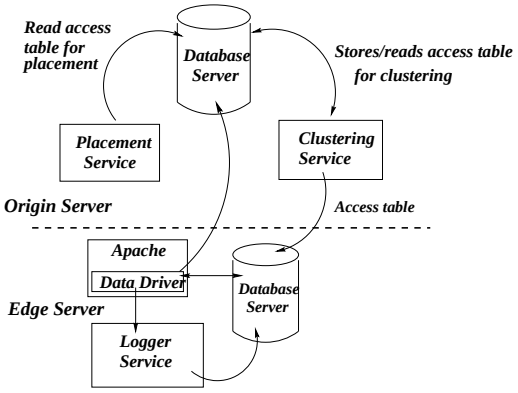


Figure 4: Salient components of the system

6.1 Implementation Overview

The salient components of our prototype system are shown in Figure 4. We implemented our data driver by modifying the existing Apache PHP driver for PostgreSQL, by adding new query interfaces to the existing PHP driver. This driver can be added as a module to the Apache Web server.

Each edge server runs a *logger service*, which is responsible for collecting information regarding which cluster was accessed by the Web clients. The logger is implemented as a stand-alone multithreaded server that collects information from the driver and periodically updates the access database.

As seen in the figure, the origin server runs two special services, *clustering* and *replica placement* services. The clustering service performs the clustering of data units during the initial stages of system deployment. It periodically collects access patterns from the edge servers and stores it in its database. Subsequently, it performs clustering with the algorithm described in Section 5. The origin server only acts a backend database so it does not need to have a Web server or logger service.

The *replica placement* service in the origin server finds the "best" locations for hosting a replica and master for each cluster. It does so by evaluating the cost obtained by different replica placement strategies and master selection for the cluster's access pattern in the previous period. Note that once data units are clustered, the logger service in the edge servers starts collecting access patterns at the cluster level. This information is then used by the replica placement service to place the replicas. Upon deciding which servers would host which clusters, the placement service builds the cluster-property table for each cluster and copies it to all edge servers.

To perform periodic adaptation, the replica placement service is invoked periodically. We note that the period of this adaptation must be set to a reasonable value to ensure good performance and stability. We intend to study the need and support for continuous adaptation and effective mechanisms for them in the future.

The clustering service is also run periodically. The bulk of its work is done during the initial stages when it has to cluster large numbers of data units. Later, during every period, it performs incremental clustering of newly created data units. The current prototype does not perform any kind of re-clustering.

6.2 Measuring the Overhead of the Data Driver

The data driver receives SQL queries issued by the application code and is responsible for locating the relevant data unit from the

local server or from a remote server. It does so by checking appropriate cluster-membership tables for each request. Performance gains in terms of latency or update traffic occur when the clusters accessed by a given edge server can be found locally. However, it is important to ensure that the gain obtained by replication is not annulled by the overhead due to checking the cluster-membership table for each data access. To analyze this, we study the response time of our driver when executing a query on a local replicated data in comparison to the response time of the original PHP driver.

In our experimental setup, we ran the Apache Web server on a PIII-900MHz Linux machine with 1 GB main memory. The PostgreSQL database also runs on an identical machine. We created a book table with fields such as book id, book name, author id, stock quantity and 5 other integer fields. The database was populated with $100,000$ random records.

We measured the execution latencies of read and write queries using the original PHP driver and the GlobeDB PHP driver for different throughput values. In both cases, the requested data is available locally; the only difference is that the GlobeDB driver needs to check its cluster membership and cluster-property tables before each data access. Read queries read a random database record using a simple query. Write queries increment the stock field of a randomly chosen book record. To make sure that each access is local, the server is assumed to be the master for all clusters. We computed the execution latency as the time taken by the server to generate the response for a request. We do not include the network latency between the client and server as the objective of this experiment is only to measure the overhead of our driver in processing the request.

The results of this experiment are given in Figure 5. As seen in the figure, even for high throughputs, the overhead introduced by our implementation is between 0 and 5 milliseconds for read accesses, and at most 10 milliseconds for writes access. This is less than 4% of the database access latencies incurred by the original driver.

We conclude that the overhead introduced by our driver is very low and, as we shall see in the next section, negligible compared to the wide-area network latency incurred by traditional non-replicated systems.

7. PERFORMANCE EVALUATION: TPC-W BOOKSTORE

The experiments presented in the previous section show that the overhead introduced by GlobeDB's driver in a single edge server is low. However, it does not offer any insight into the performance gains of GlobeDB. In this section, we study the performance gain that could be obtained using GlobeDB while hosting an e-commerce application. We chose the TPC-W benchmark and evaluated the performance of GlobeDB in comparison to other existing systems for different throughput values over an emulated wide-area network. As the experiment results presented in this section will show, GlobeDB can reduce the client access latencies for typical e-commerce applications with large mixture of reads and write operations without requiring manual configurations or performance optimizations.

7.1 Experiment Setup

We deployed our prototype across 3 identical edge servers with Pentium III 900 Mhz CPU, 1-GB of memory and 120GB IDE hard disks. The database servers for these edge servers were run on separate machines with the same configuration. Each edge server uses Apache 2.0.49 Web servers with PHP 4.3.6. We use PostgreSQL 7.3.4 as our database servers. The origin server uses an identical configuration as the edge servers except that it acts just as a backend database and does not run a Web server. We emulated a wide-area network (WAN) among servers by directing all the traffic to an intermediate router which uses the NIST Net network emulator [2]. This router delays packets sent between the different servers to simulate a realistic wide-area network. In the remaining discussion, we refer to links via NISTNet with a bandwidth of $10Mbps$ and a latency of 100ms as WAN links and $100Mbps$ as LAN links. We use two client machines to generate requests addressed to the three edge servers. A similar setup to emulate a WAN was used in [15].

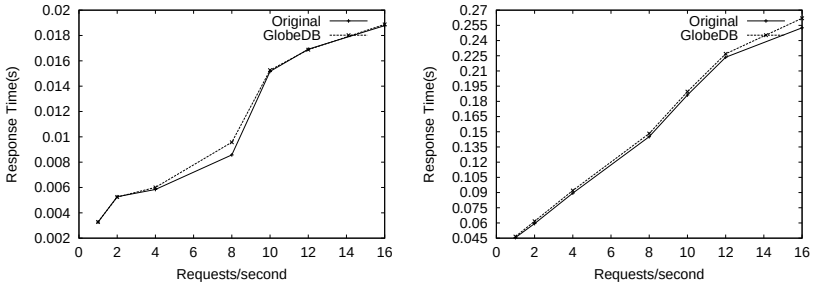
We deployed the TPC-W benchmark in the edge servers. TPC-W models an on-line bookstore and defines workloads that exercise different parts of the system such as the Web server, database server, etc. The benchmark defines activities such as multiple on-line browsing sessions using Remote Browser Emulators (RBEs), dynamic page-generation from a database, contention of database accesses and updates. The benchmark defines three different workload scenarios: browsing, shopping and ordering. The browsing scenario has mostly browsing related interactions (95%). The shopping scenario consists of 80% browsing related interactions and 20% shopping interactions. The ordering scenario contains an equal mixture of shopping and browsing interactions. In our experiments, we evaluate GlobeDB for the ordering scenario, as it generates the highest number of updates. The performance metrics of TPCW-benchmark are WIPS (Web Interactions Per Second), which denotes the throughput one could get out of a system and WIRT, which denotes the average end-to-end response time that would be experienced by a client.

The benchmark uses the following database tables: (i) $tpcw\_item$ stores details about books and their stocks; (ii) $tpcw\_customer$ stores information about the customers; (iii) $tpcw\_author$ stores the author-related information; (iv) $tpcw\_order$ and $tpcw\_orderline$ store order-related information. In our experiment, we study the effects of replication only for the $tpcw\_customer$ and the $tpcw\_item$ tables as these are the only tables that receive updates and read accesses simultaneously. The $tpcw\_author$ table is replicated everywhere as it is never updated by Web clients.

In our experiments, we want to compare the performance of GlobeDB with traditional centralized and fully replicated scenarios. In principle, a fully replicated system should replicate all tables at all edge servers. However, each record of ordering-related tables is mostly accessed by only a single customer. In addition, the entire order database is used to generate the "best sellers" page. In such a scenario, full replication of ordering-related tables would be an overkill as it would result in too much update traffic among edge servers. Any reasonable database administrator would therefore store ordering-related database records only in servers that created them and maintain a copy at the origin server (which would be responsible for generating responses to "best sellers" queries). So, for a fair comparison with GlobeDB, we implemented this optimization manually for the fully replicated system.

We use the open source PHP implementation of TPC-W benchmark [19]. We disabled the image downloading actions of RBEs as we want to only evaluate the response time of dynamic page generation of the edge server. To account for the geographical diversity of the customers, we defined 3 groups of clients which respectively issue their requests to a different edge server. We believe this is a realistic scenario as customers typically do not move often and are usually redirected to the server closest to them.

The $tpcw\_item$ table stores fields such as its unique integer identifier, author identifier, item name, price and stock. In addition to these fields, it also stores five integer fields that have identifiers of books that are closely related to it and have a similar customer



(a) Read Accesses

(b) Write Accesses

Figure 5: A comparative study of GlobeDB driver implementation with original PHP driver for reading and updating local data units

base. These fields are read by the application code to generate promotional advertisements when a client reads about a particular book. For example, related fields of a Harry Potter book may contain identifiers of the other Harry Potter books. In our experiment, we filled these related identifiers as follows: the item records are classified into three groups and related entries of each item is assigned to an item in the same group. In this way, the related field truly reflects books with similar customer base. Furthermore, TPC-W stipulates that each Web session should start with a book for promotion. In our experiments, client of edge server i receive the starting random book id only from item group i. At the outset, this might look as if each client group request books from only one item group. However, this is not the case as the RBE clients select the books to view also from other interactions, such as best-sellers and search result interactions, which have books spanning across multiple item groups.

As it can be seen, even though the assumptions we make about the clients access patterns are realistic, they do not capture all kinds of client access patterns. To address this issue, in one of our earlier studies, we simulated our proposed replication techniques for different kinds of access patterns from uniform popularity (all clients are interested in all data) to very skewed popularity (only a small set of clients are interested in a particular piece of data) using statistical distributions [25]. We found that our techniques perform well in all cases compared to traditional fully replicated and centralized systems. Hence, in this experiment we restrict our evaluations to only the workload described in this setup and study the relative performance of different system implementations.

In our experiments, for each run we ran the benchmark for 8 hours. After this, the origin server collected the access patterns from the edge servers and performed clustering and replication. Analysis of the $tpcw\_customer$ table resulted in 3 clusters. Each of these clusters were mostly accessed by only one edge server (different one in each case). Analysis of $tpcw\_item$ accesses led to 4 clusters. 3 out of the 4 clusters are characterized by accesses predominantly from only one edge server (a different one in each case). However, the fourth cluster represents data units that are accessed by clients of all the three edge servers.

7.2 Experiment Results

We evaluated the performance of four different systems (shown in Figure 6): (i) a traditional centralized architecture where the code and the database is run in a server across a WAN (Centralized), (ii) a simple edge server architecture where three edge servers run a Web server and the database is run in a server across the WAN (SES), (iii) our proposed system with 3 edge servers (GlobeDB)

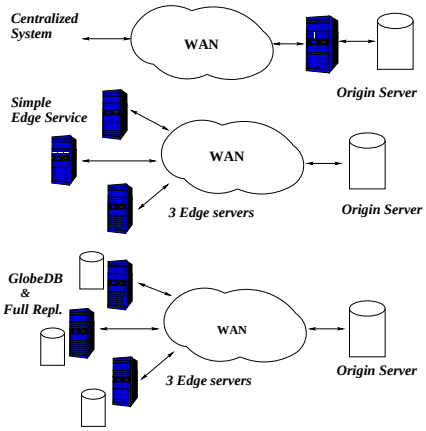


Figure 6: Architectures of different systems evaluated

and (iv) a full replication system (Full) which is similar to the GlobeDB setup - the only difference being that the $tpcw\_item$ and $tpcw\_customer$ tables are fully replicated at all edge servers unlike GlobeDB.

As we noted earlier, replication decisions are made through evaluation of the cost function and its weights α, β and γ as described in Section 5. In our experiments, we assumed the system administrator wants to optimize the system for improved response time and assigned higher weights to α and β compared to γ. We set α=$2/r_{max}$; β=$2/w_{max}$; and γ=$1/b_{max}$, where r_{max}, w_{max} and b_{max} are maximum values of average read latency, write latency, and number of consistency updates, respectively. These values effectively tell the system to consider client read and write latency to be twice as important in comparison to update bandwidth (on a normalized scale). These weights result in the following placement configuration: For the 3 clusters of the customer table, 3 different placement configurations are obtained where a different edge server is the master while hosting a replica of the cluster. Similar placement configurations are obtained for 3 out of the 4 clusters of the book table. The fourth cluster (which contains data units that were accessed from all edge servers) was automatically placed at all the 3 edge servers as it represents book records that are popular among all clients.

In our experiments, we study the WIRT for different WIPS until the database server cannot handle more connections. The results of our experiments are given in Figure 7. As seen in the figure, even for low throughputs GlobeDB performs better than the traditional Centralized and SES architectures and reduces response time by

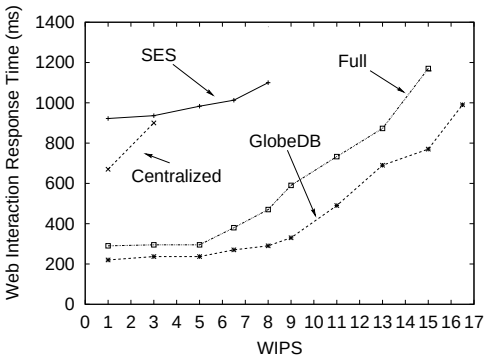


Figure 7: Performance of different system architectures running TPC-W benchmark

a factor of 4. GlobeDB performs better than SES and centralized system as it is able to save on wide-area network latency for each data access as it is able to find many data records locally. Moreover, this also shows that replicating application code is not sufficient to obtain good performance.

The difference in WIRT between the GlobeDB and Full setup varies from 100 to 400ms. This is because the GlobeDB system is capable of performing local updates (the server that writes most to a database cluster is elected as its master) but the Full setup forwards all updates to the origin server. These updates constitute 30% of the workload. On the other hand, the Full setup gains in the fact it can handle some complex queries such as search result interactions locally, while GlobeDB forwards it to the origin server.

It is obvious that the fully replicated system produces more update traffic than GlobeDB as it propagates each update to all edge servers. In this experiment, we found that GlobeDB reduces the update traffic by a factor of 6 compared to Full replication. This is because GlobeDB prevents unnecessary update traffic by placing data only where they are accessed. Reducing the update traffic potentially leads to significant cost reduction as CDNs are usually charged by the data centers for the amount of bandwidth consumed.

The centralized and SES systems pay the penalty for making connections over a WAN link. Note that between these systems the Centralized setup yields lower response time as each client request travels only once over the WAN link. In the SES setup, each Web request triggers multiple data accesses that need to travel across WAN links thereby suffering huge delays. However, among these traditional systems, the SES architecture yields better throughput as it uses more hardware resources, i.e., the Web server and the database system are running in separate machines. SES yields a throughput of 7.9 req/s, while the centralized architecture yields much less (1.9 req/s) as it runs both Web server and database on the same machine.

Replication also affects throughput. GlobeDB attains a throughput of 16.9 req/sec and is 2 WIPS better than the Full setup and 8 WIPS better than SES. It performs better than SES because the data handling workload is shared among many database servers. While the Full setup has the same number of database servers as GlobeDB, the latter sustains higher throughput as a server does not receive updates to data that it accesses rarely. This reduces the workload of individual database servers in addition to reducing overall consumed bandwidth.

In conclusion, the results show that the GlobeDB's automatic data placement can reduce client access latencies for typical e-commerce applications with a large mixture of read and write operations. With these experiments we have shown that automatic placement of application data is possible with very little administration. We believe this is a promising start for realizing a truly scalable self-managing platform for large-scale Web applications.

7.3 Effect of the Cost Function

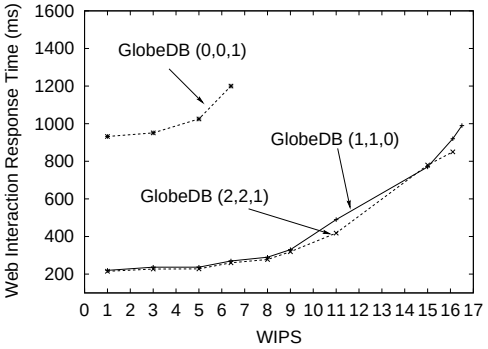


Figure 8: Relative performance of autonomic system architectures

In our earlier experiments, we showed that autonomic placement of data can yield better performance than traditional strategies. As we noted in Section 5, the underlying decision making tool used for autonomic placement is the *cost function* and its weights α, β, and γ.

The objective of the results presented in this section is not to show what are the right weights for a system. Rather, this experiment is just a guide to show the utility of our cost function and the simplicity with which it can attain different objectives (latency optimization, bandwidth optimization or a mixture of it).

We evaluated the relative performance of autonomic systems that uses three different weight parameters, in effect three different criteria for placement. The three systems evaluated are: (i) GlobeDB (0, 0, 1): a system with weights (α, β, γ) =(0,0,1), which implies the system wants to preserve only the bandwidth and does not care about latency, (ii) GlobeDB (1, 1, 0): a system whose weights are set such that the system cares only about the client latency and does not have any constraints on the amount of update bandwidth. Effectively, this situation leads to creating more replicas. (iii) GlobeDB (2, 2, 1): a system that prefers to optimize latency twice as much as the update bandwidth.

As can be seen, these systems are designed for different conditions. For example, GlobeDB(0, 0, 1) is useful for a system that cannot afford to pay for the wide-area bandwidth consumed in its remote edge servers and GlobeDB(1, 1, 0) is useful for a CDN that values its client quality of service (QoS) more than its maintenance costs consumed for maintaining consistency among replicas.

The goal of this experiment is to analyze the impact of different cost function parameters on the client response time of the TPC-W benchmark and the results are given in Figure 8. As seen in the figure, GlobeDB(0, 0, 1) performs the worst in terms of WIRT, as it leads to a placement where the data are not replicated at all. Furthermore, its throughput is saturated because all transactions are sent to a single database server. The other two systems perform equally well and yield good throughput. With respect to update traffic, GlobeDB(2, 2, 1) performs better than GlobeDB(1, 1, 0) and reduces update traffic by a factor of 3.5. Note that, while GlobeDB(1, 1, 0) and a fully replicated system have similar goals, the former yields better WIRT as it is able to perform local updates.

8. RELATED WORK

A number of systems have been developed to handle Web application replication [4, 7, 22]. These systems replicate code at the

replica servers, but either do not replicate the application data or cache them at the replica servers. Furthermore, all these systems also rely on manual administration and the administrator must decide which data must be cached and placed at which server. This is the most important shortcoming of existing systems.

Commercial database caching systems such as DBCache [6] and MTCache [17] cache the results of selected queries and keep them consistent with the underlying database. Such approaches offer performance gains provided the data accesses contain few unique read and/or write requests. However, the success of these schemes depends on the ability of the database administrator to identify the right set of queries to cache. This requires a careful manual analysis of the data access patterns to be done periodically.

In [15], the authors propose an application-specific edge service architecture, where the application itself is supposed to take care of its own replication. In such a system, access to the shared data is abstracted by object interfaces. This system aims to achieve scalability by using weaker consistency models tailored to the application. However, this requires the application developer to be aware of an application's consistency and distribution semantics and to handle replication accordingly. This is in conflict with our primary design constraint of keeping the process of application development simple. Moreover, we demonstrated that such an awareness need not be required, as distribution of data can be handled automatically.

Recently, database researchers have built middleware systems such as C-JDBC [8] and Ganymed [21] for scalable replication of a database in a cluster of servers. However, the focus of these works is to improve the throughput of the underlying backend database within a cluster environment, while the focus of our work is to improve the client-perceived performance and reducing wide-area update traffic. We also note that these works can be combined with GlobeDB to scale the database in a single edge server.

9. CONCLUSION

In this paper, we presented GlobeDB, a system for hosting Web applications that performs efficient autonomic replication of application data. We presented the design and implementation of our system and its performance. The goal of GlobeDB is to provide data-intensive Web applications the same advantages CDNs offered to static Web pages: low latency and reduced update traffic. We demonstrated this with experimental evaluations of our prototype implementation running the TPC-W benchmark over an emulated wide-area network. In our evaluations, we found that GlobeDB significantly improves access latencies and reduces update traffic by a factor of 6 compared to a fully replicated system. The major contribution of our work is to show that the process of application development can be largely automated and in such a way that it yields substantial improvement in performance.

We note that our current prototype assumes that the system is free of server and network failures. However, such failures may create a consistency problem if the master for a data unit is unreachable. We plan to address these issues in the near future.

The system presented in this paper is the first step in achieving the goal of complete autonomic system for replication. There are several open issues that need to be addressed to realize an autonomic CDN we envisage. These issues include effective re-clustering, continuous adaptation of placement configurations and fault tolerance. We plan to address them in our next steps.

10. ACKNOWLEDGEMENTS

We thank Christian Plattner (ETH) and Arno Bakker (VU) for their helpful discussions and helping us with the setup of the emulator test bed. We also thank the anonymous reviewers for their comments that helped in improving the clarity of the paper.

11. REFERENCES

[1] Java TPC-W Implementation Distribution, `http://www.ece.wisc.edu/~pharm/tpcw.shtml`.

[2] NISTNet: A Network emulation tool, `http://snad.ncsl.nist.gov/itg/nistnet/`.

[3] PostgreSQL, `http://www.postgresql.org/`.

[4] Akamai Inc. Edge Computing Infrastructure.

[5] B. H. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

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

[7] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the Web. In *Proceedings of the Middleware Conference*, pages 373–388, Sept. 1998.

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

[9] J. Challenger, P. Dantzig, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, 4(4), Nov 2004.

[10] Y. Chen, L. Qiu, W. Chen, L. Nguyen, and R. H. Katz. Clustering web content for efficient replication. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP'02)*, pages 165–174, 2002.

[11] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 97–108. ACM Press, 2002.

[12] 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.

[13] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[14] 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 *Proceedings of INFOCOM*, pages 783–791, March 1998.

[15] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proceedings of the Twelfth International World Wide Web Conference*, pages 449–460, May 2003.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, San Mateo, CA., 1993.

[17] P. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL server. *Data Engineering*, 27(2):27–33, June 2004.

[18] W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, and D. Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proceedings of the Twelfth International Conference on World Wide Web*, pages 587–598. ACM Press, 2003.

[19] PGFoundry. PHP Implementation of TPC-W Benchmark, `http://pgfoundry.org/projects/tpc-w-php/`.

[20] G. Pierre, M. van Steen, and A. S. Tanenbaum. Dynamically selecting optimal distribution strategies for Web documents. *IEEE Transactions on Computers*, 51(6):637–651, June 2002.

[21] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, Toronto, Canada, Oct. 2004.

[22] M. Rabinovich, Z. Xiao, and A. Agarwal. Computing on the edge: A platform for replicating internet applications. In *Proceedings of the Eighth International Workshop on Web Content Caching and Distribution*, pages 57–77, Hawthorne, NY, USA, Sept. 2003.

[23] M. Rabinovich, Z. Xiao, F. Douglis, and C. Kalmanek. Moving edge-side includes to the real edge - the clients. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[24] S. Sivasubramanian, G. Pierre, and M. van Steen. A case for dynamic selection of replication and caching strategies. In *Proceedings of the Eighth International Workshop Web Content Caching and Distribution*, pages 275–282, Hawthorne, NY, USA, Sept. 2003.

[25] S. Sivasubramanian, G. Pierre, and M. van Steen. Scalable strong consistency for web applications. In *Proceedings of ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[26] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen. Replication for web hosting systems. *ACM Computing Surveys*, 36(3), Sept. 2004.

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

[28] Speedera Inc. `http://www.speedera.com`.

[29] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welsh. Session Guarantees for Weakly Consistent Replicated Data. In *3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, Sept. 1994. IEEE.