Scalable Strong Consistency for Web Applications

Swaminathan Sivasubramanian Guillaume Pierre Maarten van Steen
Dept. of Computer Science, Vrije Universiteit
Amsterdam, The Netherlands
{swami,gpierre,steen}@cs.vu.nl

Abstract

Web application workloads are often characterized by a large number of unique read requests and a significant fraction of write requests. Hosting these applications drives the need for the next generation CDN architecture that does more than caching the results of Web applications but replicates both the application code and its underlying data. We propose the design of a system that guarantees *strong consistency* for Web applications with high *scalability*. The proposed system is based on partial replication, where data units are replicated only to servers that access them often. This reduces the consistency overhead as updates are sent to a reduced number of servers. The novelty of our system is that the proposed partial replication is performed by the system *automatically* by analyzing the system's access patterns periodically. We explore the design space of this system, find the key issues that need to be addressed to build it and propose solutions to solve them. We further show that the proposed algorithms offer significant performance gains compared to existing solutions for a wide range of Web access patterns.

1 Introduction

A growing number of e-commerce applications can be characterized by a large number of unique read requests and a significant fraction of write requests. Hosting these applications in a centralized server (or cluster of servers) may result in poor response time for Web clients due to wide-area network latency introduced for each access. To improve their performance, many systems cache the pages generated by the applications. However, such solutions rely on the assumptions that the temporal locality of requests is high and the updates are infrequent. Applications that do not exhibit these characteristics can only be distributed using replication, where the application code is executed at the replica servers. This avoids the wide-area network latency for each read/write access and ensures quicker response time to clients.

Replicating a Web application requires replicating both the application code (e.g., EJBs, CGI scripts, PHPs) and the data that the code acts upon (databases or files). This is relatively easy provided that the code does not modify the data [11]. However, most applications do modify their underlying data. In this case, it becomes necessary to manage data consistency across all replicas. Efficient replication of such applications on a worldwide scale is difficult because it implies significant replica update traffic or high write access latencies. This bottleneck can be avoided by adopting weak consistency models, which, in turn, requires significant expertise from the application developers. In this paper, we focus on scalable solutions to guarantee strong consistency for Web applications.

We explore an approach based on partial data replication, which we call on-demand replication. Data is segmented into data units and each data unit is replicated only to servers that access it frequently. So, the entire data set is not replicated at all replica servers. This approach can reduce the synchronization overhead as consistency updates for a data unit are sent to a reduced number of servers.

We believe that on-demand application replication is useful for general e-commerce applications, as it allows the system to exploit the location-specific interests in request patterns. For instance, a worldwide e-commerce application does not need to replicate its customer database to all its replicas. North American customer records can be stored primarily in replica servers in North America and need not be replicated to Asian servers. Though storage is not an issue with sharp decline in storage costs, the synchronization costs would then be reduced when a customer record is updated.

Although we believe that data segmentation can help to replicate Web applications, it may be difficult for application developers or system administrators to come up with efficient schemes. We therefore propose that data segmentation and replication to be performed *automatically* based on their access patterns.

Building a system for on-demand application replication requires addressing many issues such as identifying the granularity and constituents of the data segments, finding the optimal placements for each data segment and the code, managing partially replicated data, and choosing the optimal consistency strategy for each data segment. The contributions of this paper are as follows: (i) We explore the design space of such a system and identify some of the key issues that one needs to address to realize such a system and suggest solutions to solve them, and (ii) we also show that such on-demand replication can provide significant performance gains.

The rest of the paper is organized as follows: Section 2 presents our application model. Sections 3 and 4 respectively discuss data clustering and replication techniques. Section 5 evaluates the performance gains due to on-demand replication. Finally, Section 6 discusses the related work and Section 7 concludes the paper.

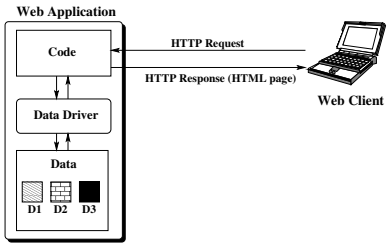


Figure 1: Application Model

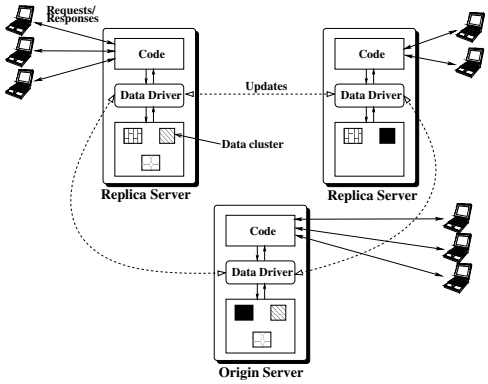


Figure 2: System Architecture

2 System Model

2.1 Application Model

An important issue when replicating Web data is to decide to which extent the applications should be aware of replication. Replication can yield the best performance if it is completely tuned to the specific application and its access patterns. However, this requires significant effort and expertise from an application developer, for which reason optimal performance of the application is often not reached. Furthermore, changes in access pattern may warrant changes in replication strategies. This makes the process of developing an optimal replication strategy for the application next to impossible.

In our system, we made the opposite choice by having a completely *replication-transparent* application model. In our model, the application developer need not worry about replication issues but only stick to functional issues. The system will automatically derive a replication strategy for a given application, under possibly varying access patterns.

For the same reasons, we decided that our system should provide sequential consistency. This consistency model enables the application developer to develop applications as if the underlying data were concurrently accessed from a centralized location, thereby ignoring distribution issues.

Our application model is given in Figure 1. As seen in the figure, an application is made of code and data. The code is written using standard technologies such as Active Server Pages (ASPs), CGI scripts or EJBs deployed in an application server. The code receives HTTP requests from its Web clients and issues read/write accesses to the relevant data (in a database or filesystem) to answer them.

Access to the data is realized by a data driver, which acts as the interface between the code and data. It preserves distribution transparency of the data as it hides the fact that data are partially replicated. The data driver has a simple file-system-like interface (for accessing file-based data) or JDBC-like interfaces (for databases) and is responsible for finding the data required by the code, either locally or from a remote server.

We assume that the data are 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. Examples of data units are files, database tables, or even records. The system replicates each data unit according to its specific pattern.

Choosing the right granularity of data for replication has important performance implications. If the granularity is too coarse, we may lose the benefits of partial replication. On the other hand, if it is too fine, the overhead for handling replicas will be high. 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 automatically grouped by the system into a single cluster. The system subsequently handles replication at the cluster level, thereby making the problem of tracking a cluster tractable without losing the benefits of partial replication. Clustering is described in more details in Section 3.

2.2 System Architecture

The architecture of our proposed system is presented in Figure 2. A given application is hosted over m replica servers spread across the Internet. As seen in the figure, the application data are partially replicated, so each server hosts only a subset of all data clusters. We assume that Web clients are redirected to their closest replica server using standard technologies, such as DNS-based redirection.

According to an application's access pattern, the system must cluster data units and decide on the assignment of a replication strategy for each cluster. 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 servers. The origin server also acts as the initial replica for all data clusters.

As noted earlier, we want to provide replication transparency for which we believe sequential consistency is the best model. This choice requires an efficient consistency protocol in the presence of concurrent updates. We adopted a master-slave consistency protocol: each data cluster has a master server responsible for synchronizing simultaneous updates emerging from different replica. The master for each data cluster is selected by the origin server. Read requests for a data cluster are forwarded to the closest server that contains a replica (if not present at the replica server that received the client request). Write requests for a data cluster are always forwarded to the master. The master pushes the update to its slaves each time the contents of the data cluster is modified. Issuing all write operations to a given cluster at a single location effectively serializes updates, which generates sequential consistency. We chose the protocol primarily for its simplicity and because is is sufficient for clarifying our position.

3 Data Clustering

As discussed before, fine-grained data segmentation introduces a large number of individual data units, posing a scalability problem for replication algorithms. We propose to cluster data units with similar usage patterns and replicate data at the cluster level instead of the data unit level. Subsequent cluster membership can be efficiently handled through bit arrays and Bloom filters [3].

Our system consists of m replica servers R_1, R_2,...,R_m, holding data units D_1,D_2,...,D_n. We want to group data units with similar read and write access patterns. However, for the sake of simplicity, we limit our discussion to only read access patterns. The techniques presented here can be easily extended to accommodate write access patterns.

Each data unit D_i has an access pattern A_i=$\{r_{i,1}$,$r_{i,2}$,...,$r_{i,m}\}$, where $r_{i,j}$ is the number of read accesses made by the replica server R_j for a data unit D_i. We want to group two data units D_i and D_j into the same cluster, if A_i and A_j are similar.

A similar problem of clustering has been studied before in the context of collaborative filtering or recommender systems [7]. Recommender systems observe the access patterns of end-users to products, and try to cluster end-users who have similar interests. This allows the system to issue personalized recommendations for products that can be of interest to each particular user. This problem is similar to ours, where the role of end users is played by replica servers and the role of products is played by the data units.

A similarity metric frequently used in recommender systems is cosine-based similarity. It considers the access patterns of data units as m-dimensional vectors. Similarity between two data units D_i and D_j is given by

$$Sim(i,j) = cos(A_i, A_j) = \frac{\overrightarrow{A_i}.\overrightarrow{A_j}}{\|\overrightarrow{A_i}\|\|\overrightarrow{A_i}\|}.$$

Data units D_i and D_j are clustered if $Sim(i,j) \geq 1 - x$, where x is a threshold such that $0 \leq x \leq 1$.

In general, clustering a large number of data units is computationally expensive, of order $O(m * n)$. However, since the access vectors of data units are sparse in nature, it can effectively be reduced to $O(m + n)$.

Another important step is to handle the run-time creation of a new data unit. In our system, new data units are always created at the origin server. They are not replicated immediately: the origin server first collects the access pattern for the new data unit and determines which cluster should hold it. If no suitable cluster is found, the origin server creates a new cluster containing the new data unit. A new data unit is replicated as soon as it is inserted into a cluster.

4 Replication Policies

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 discuss algorithms concerning the selection of the "best" replication strategy for a data cluster.

A replication strategy describes at least three aspects: *replica placement*, *consistency mechanism*, and, in our case, *master selection*. Placement mechanisms dictate the number and location replicas, while consistency mechanisms dictate the protocol used to enforce data consistency among replicas. Master selection mechanisms decide on the master replica responsible for handling concurrent updates for a data cluster. As we made the choice of a master-slave consistency protocol, the selection of the "best" replication strategy involves deciding only on replica placement and master selection.

To select the "best" replication strategy for a data cluster, the system needs to know what the definition of "best" performance is. One can measure the performance of the system with a number of metrics such as the average read latency, the average write latency, the amount of update traffic, etc. However, optimizing the system performance for one of these metrics alone would often result in degrading the others. 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.

It can be seen that there is a clear tradeoff between the performance gain due to replication and the performance loss due to its consistency enforcement. In our system, we propose to represent the overall system performance into a single abstract figure using a *cost function*. An example of a cost function that measures 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, b is the amount of bandwidth used for consistency enforcement, and α, β and γ are weights associated to each metric. Weights must be set by the system administrator based on the system constraints and application requirements. A larger weight implies that its associated metric has more influence in selecting the "best" strategy. Finding the "best" system configuration now boils down to evaluating the value of the cost function for every candidate strategy. By definition, the best configuration is the one with the lowest cost.

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 our system, we employ the use of heuristics to perform replica placement and master selection. For each problem, we propose a number of possible heuristics. This reduces the problem of choosing replication strategy to evaluating which combination of heuristics performs the best in any given situation.

4.1 Replica Placement Heuristics

In our system, the origin server periodically collects the access patterns of each data cluster from all replica servers. Subsequently, it places a replica of a data cluster in a replica server, if it generates at least $x\%$ of data access requests. This creates a family of heuristics Px.

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, therefore to lose the advantages of replication. On the other hand, a low value of x will lead to a fully replicated configuration, losing the advantages of partial replication. Hence, it is important to choose the right value of x based on the access patterns of the data cluster.

Expecting the system administrator to determine x is not reasonable, as the number of parameters that affect the performance is high. Instead, administrators are just expected to define their preferred performance tradeoffs by choosing the weight parameters of the cost function. The system will automatically adjust the replication configuration to the one that gives the best cost.

4.2 Master Selection Heuristics

Careful master selection is essential to optimize the write latency and the amount of bandwidth utilized to maintain consistency among replicas. In our system, we consider two heuristics for master selection mechanisms. The *most-writer* heuristic selects the master as the replica server that generates the highest number of update requests. This strategy allows the highest fraction of write requests to be handled locally. However, if all servers issue similar numbers of write requests, this strategy will give poor write latency because a large fraction of write requests will be redirected to the master, which is not necessarily topologically close to the other writers.

This problem is avoided by the *closest-writer* heuristic, which selects the server that offers the least average write latency as the master. If n_i is the number of write requests by replica server R_i and l_{ij} is the latency between replica server i and j, then the average write latency for a data cluster whose master is k is given by: $w_k = (\sum_{i=1}^{m} n_i * l_{ik})/(\sum_{i=1}^{m} n_i)$. The *closest-writer* heuristic selects the server with lowest average write latency as the master.

5 Performance Evaluation

In this section, we show that on-demand data replication can provide considerable performance gains. Moreover, depending on the access patterns, different policies perform best. This demonstrates that our system should dynamically adapt its policies when the access patterns change.

Ideally, we would perform experiments based on real-world traces of data accesses from a global dynamic Web site. However, the lack of such publicly available traces restricted us to simulating the system with synthetic traces.

Building a fair experiment that simulates an Internet-wide CDN has two important challenges: (i) simulating a wide-area network with realistic network delays between servers placed worldwide; and (ii) simulating the diversity of interest among clients in each particular piece of data.

To simulate a set of servers to host a Web application, we selected 100 hosts that visited our department Web site, such that they are spread across 6 continents and 66 countires, and can represent our replica servers. We estimated the latencies between each pair of hosts using SCOLE [14]. SCOLE associates hosts with co-ordinates in an N-dimensional space by measuring their latency to a fixed number of known landmarks. The latency between two positioned hosts is calculated as the Euclidian distance of their co-ordinates in this space. This method of latency estimation is shown to be fairly accurate while requiring relatively few measurements. Latencies between our servers range from 23 ms to 2700 ms.

Simulating the diversity of client interests for a particular data cluster is harder to solve. This is an important factor as the diversity of client interest influences the performance of on-demand policies. For example, if a data item is of interest to only a small subset of servers, then on-demand replication can give huge performance gains in terms of read/write latency and update traffic. On the other hand, if the data is of equal interest to clients of all replica servers, then the performance gains due to on-demand replication will not be high as the data is required everywhere. However, it is important for us to study the performance of on-demand replication in the full spectrum of cases.

We modelled the diversity of client interest using statistical distributions. Since, to our knowledge, there is no earlier study that has modelled the geographical influence of client requests to a database, we based our simulations on a Zipf distribution for generating diversity in client interest for a particular data cluster. We take the exponent a in the Zipf distribution as an input parameter for our experiments.[1] To study our system in diverse access patterns, we vary the parameter a. A trace generated with a low value of a implies that each server is equally likely to access a data cluster i.e., the distribution of client interest is flat. In contrast, a high value of a generates a trace where each data cluster is of interest only to a small number of servers (those with the highest rank) i.e., the distribution of client interest is more skewed. In our simulations we varied a between 0 and 3.

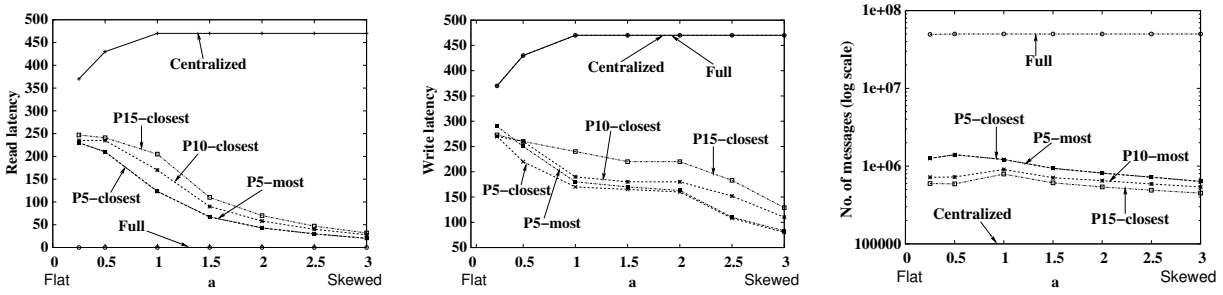
We measure the system performance using the following metrics: (i) **Average Read latency (ARL):** the average latency incurred by read requests for a data cluster, (ii) **Average Write latency (AWL):** the average latency incurred by write requests for a data cluster and (iii) **Number of consistency messages (NCM):** the number of update messages sent among replica servers to keep the data consistent (excluding the client-to-replica traffic). NCM serves as an indicator of the amount of bandwidth utilized by the system just for maintaining data consistency.

Each simulation consists of $1,000,000$ requests, half of which are write requests. We study the performance of the following policies: (i) centralized solution, (ii) fully replicated solution with origin server as the master (Full), (iii) $P5 - closest$, (iv) $P10 - closest$, (v) $P15 - closest$ and (vi) $P5 - most$. To be fair on the centralized and fully replicated solutions, we chose the best possible replica server as the origin server. That is, when assuming that all servers have the same access pattern, the server with the minimum average latency to other servers is chosen as the origin server.

Due to the lack of space, we present only some of our simulation results. For a detailed performance evaluation, please refer to [12].

Figure 3 presents the effect of varying the value of a on the system performance. As can be seen, on-demand replication produces significant gain in terms of read/write latencies (by

[1] A Zipf distribution states that the frequency of occurrence of a particular value i is given by $f_i = C \cdot r_i^{-a}$ where r_i is the rank of i's occurence.



(a) Effect of a on the Average Read Latency

(b) Effect of a on the Average Write Latency

(c) Effect of a on the Number of Consistency Messages

Figure 3: Effect of distribution skew on system performance

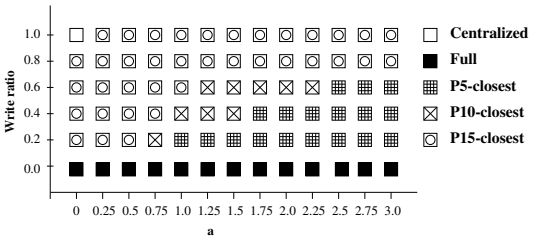


Figure 4: Best Replication Policy for Different Access Patterns

a factor of 4) and reduced update traffic (by two orders of magnitude) compared to fully replicated or centralized systems. The more the client diversity increases, the best our system performs. However, even in the case of a flat distribution of interests (low values of a), on-demand replication policies give low write latencies and reduced update traffic.

It must be noted that gaining two orders of magnitude in network traffic is of immense significance to a worldwide CDN, as the Internet is often affected by network congestion. Reduced number of consistency messages also will lead to improved write latency of the system and less cost, as CDNs are usually charged by ISPs and data centers based on the amount of traffic they generate.

We now address the question: which policy performs best for a given access pattern? In the following experiment, we vary both a and the ratio of write requests among the total 1,000,000 requests. For each such defined access pattern, we simulated each replication policy and selected the best one as the one with the lowest cost. We normalized the weights of the cost function such that each parameter has roughly equal significance: α=$1/r_{max}$; β=$1/w_{max}$; and γ=b_{max}, where r_{max}, w_{max} and b_{max} are maximum values of average read latency, write latency, and number of consistency messages respectively. Figure 4 shows which policy performs best for each request pattern.

As seen in the figure, depending on the access pattern, different policies perform best. For example, an application with no updates (write-ratio=0) performs best with full replication, as all requests will then be served locally. Similarly, if there is a flat distribution of clients and only write requests ($a = 0$ and write-ratio=1), the centralized solution performs the best as it has a replica only at the origin server thereby giving the best average write latency without any update traffic overhead (note that the origin server was selected as the server that gives the least average latency). For all other values, on-demand replication performs best. Policies with higher threshold perform best when the request distribution is flat, as in such cases placing less replicas yields reduced update traffic. On the other hand, when a small number of servers generate most of the requests (be it reads, writes, or a combination thereof) it is preferable to place more replicas, and each close to where the requests come from.

This result also suggests that replication policies should be selected on a per-data cluster basis according to their access patterns. We propose that our system periodically evaluates the cost of different policies for each data cluster. The system can then dynamically adapt its policies on a per-cluster basis to provide optimal performance.

6 Related Work

For the past decade, numerous solutions have been proposed in the context of caching systems for delivering Web content [13]. Most of these systems assume that the temporal locality of the client requests is high, as these systems were initially built for delivering static Web documents. Systems such as Akamai's ESI and WebViews [9] rely on this assumption and cache the results of the Web application. Unfortunately, this assumption is often not true for applications characterized by a large number of unique reads or a significant number of writes. Such applications can be distributed only using replication (of both code and data), where the application code is executed at the replica servers.

A number of systems have been developed to handle Web application replication [1, 2, 5]. These systems replicate the code at the replica servers, but either do not replicate the application data or cache them at the replica servers. This limits the system performance as all write requests need to be forwarded to a single remote location.

In [6] 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 and each replica communicates to another using a persistent messaging layer. This system aims to achieve scalability by using weaker consistency models that suits the application. However, this requires the application developer to be aware of an application's consistency and distribution semantics so that this knowledge can be used while developing these objects. This is in conflict with our primary design constraint of keeping the process of application development simple.

Our work has strong ties to partitioning in distributed databases [10], a distinction being that fragments are usu-

ally not created based on runtime analysis of access patterns. Partitioning traditionally leads to problems when applications need different fragments of the same relation, such as in a join. In our approach, we expect to circumvent this problem through proper data clustering by first choosing a fine granularity (e.g., a single row in a relation) and subsequently clustering rows into fragments based on actual access patterns. However, further research is needed to substantiate our claim of scalability for real Web applications.

Recently, database sesearchers have built systems such as DB-Cache [4] and MTCache [8], which cache the results of selected queries and keep them consistent with the underlying database. Such approaches will offer performance gains provided that the access patterns contain few unique read and 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 careful manual analysis of the data access patterns to be done periodically to identify the current "hot" requests.

7 Conclusions and Future Work

This paper explores the design space of a scalable Web application replication system that guarantees strong consistency. We adopt a simple application model for the system, which we expect will ease the process of application development. The novelty of our approach is that it employs partial replication where the data is dynamically replicated only to servers that access them. This allows the system to exploit location-specific interests in request patterns.

We show that on-demand replication performs better than centralized and fully replicated systems by reducing the average latency of read/write data access as well as the amount of traffic to maintain replicas consistent. Moreover, the best replication strategy depends on the data cluster's access pattern. Our system will automatically select the best replication strategy for a given situation through run-time evaluation of a cost function.

We have implemented the proposed system as a PHP driver for PostgreSQL database and are in the process of measuring its performance.

The proposed cluster-based replication is suitable for large scale databases where the access patterns of individual data units do not change often. However, if they do, then it is necessary to re-cluster each cluster periodically to ensure that a cluster does not contain data units with different access patterns. To avoid the problem of periodic re-clustering, we plan to explore the design of fine-grained data replication that performs data replication at fine-grained level.

References

[1] AKAMAI INC. Edge Computing Infrastructure.

[2] AWADALLAH, A., AND ROSENBLUM, M. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proc. of the Seventh International Workshop on Web Content Caching and Distribution* (Aug. 2002).

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

[4] BORNHVD, C., ALTINEL, M., MOHAN, C., PIRAHESH, H., AND REINWALD, B. Adaptive database caching with DBCache. *Data Engineering 27*, 2 (June 2004), 11–18.

[5] CAO, P., ZHANG, J., AND BEACH, K. Active cache: Caching dynamic contents on the Web. In *Proc. of the Middleware Conference* (Sept. 1998), pp. 373–388.

[6] GAO, L., DAHLIN, M., NAYATE, A., ZHENG, J., AND IYENGAR, A. Application specific data replication for edge services. In *Proc. of the Twelfth International World-Wide Web Conference* (2003), pp. 449–460.

[7] HERLOCKER, J. L., KONSTAN, J. A., BORCHERS, A., AND RIEDL, J. An algorithmic framework for performing collaborative filtering. In *Proc. of the 22nd ACM SIGIR conference on Research and development in information retrieval* (1999), pp. 230–237.

[8] KE LARSON, P., GOLDSTEIN, J., GUO, H., AND ZHOU, J. MTCache: Mid-tier database caching for sql server. *Data Engineering 27*, 2 (June 2004), 27–33.

[9] LABRINIDIS, A., AND ROUSSOPOULOS, N. Webview materialization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (2000), ACM Press, pp. 367–378.

[10] OZSU, T., AND VALDURIEZ, P. *Principles of Distributed Database Systems*, 2nd ed. Prentice Hall, Upper Saddle River, N.J., 1999.

[11] RABINOVICH, M., XIAO, Z., AND AGARWAL, A. Computing on the edge: A platform for replicating internet applications. In *Proc. of the Eighth International Workshop on Web Content Caching and Distribution* (Hawthorne, NY, USA, Sept. 2003).

[12] SIVASUBRAMANIAN, S., PIERRE, G., AND VAN STEEN, M. A system for on-demand Web application replication, Dec. 2003. `http://www.globule.org/`.

[13] SIVASUBRAMANIAN, S., SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. Web replica hosting systems. Tech. Rep. IR-CS-001, Vrije Universiteit, Amsterdam, The Netherlands, May 2003.

[14] SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. Scalable cooperative latency estimation. In *Proceedings of the 10th International Conference on Parallel and Distributed Systems (ICPADS)* (Newport Beach, CA, USA, July 2004).