Towards On-Demand Web Application Replication

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

ABSTRACT

The increasing popularity of Web applications drives the need for systems that offer improve high performance web services by replicating applications at geographically distributed servers. In this paper, we propose a system for Web application replication that replicates both application code and its data. A key challenge in such a system is data replication and consistency, so that each replica server can access its shared data locally thereby avoiding the wide-area network latency that would be incurred by a traditional centralized database. In our system, we aim to minimize this consistency overhead using partial replication where data units are replicated to servers that only access them. This would reduce the consistency overhead as updates are sent only to servers that access them. 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.

Categories and Subject Descriptors

C.2 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems; C.4 [**PERFORMANCE OF SYSTEMS**]: Design studies and serviceability

Keywords

Web Application Replication, Distributed Web Services

1. INTRODUCTION

A growing number of e-commerce applications can be characterized by a large number of unique read requests and a significant write-to-read ratio. 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 [19].

However, most applications do modify their underlying data. In this case, it becomes necessary to manage data consistency across all replicas. As suggested in [5, 11], the core challenge in this problem is data distribution and consistency. With a significant number of updates to the data replicated across multiple servers in the Internet, the overhead involved in maintaining the consistency of these replicas (in terms of network bandwidth and write access latency) becomes high.

Reducing consistency overhead in replicated systems is often realized by employing weak consistency models [7, 23]. However, weak consistency mechanisms (with the exception of time-based mechanisms) require the application developer to understand the distribution and consistency semantics of the application, which complicates the process of application development.

In this paper, we explore another 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 only to replica servers that access them often.

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 to come up with efficient schemes. We therefore propose that data segmentation across replicas 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. This paper explores the design space of such a system. We identify some of the key issues that one needs to address to realize such a system and suggest solutions to solve them.

The rest of the paper is organized as follows: Section 2 presents the application model for the Web applications used in our system. Section 3 and 4 respectively discuss our data clustering and replication techniques. Section 5 discusses the related work and Section 6 concludes the paper.

Copyright is held by the author/owner(s).
WWW2004, May 17–22, 2004, New York, NY USA.
.

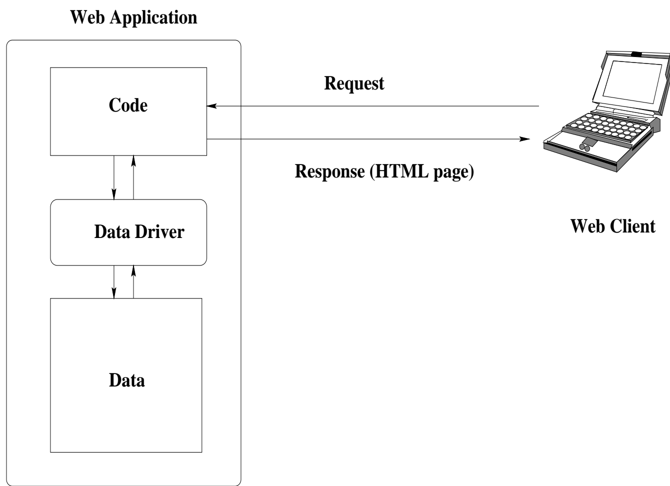

Figure 1: Application Model

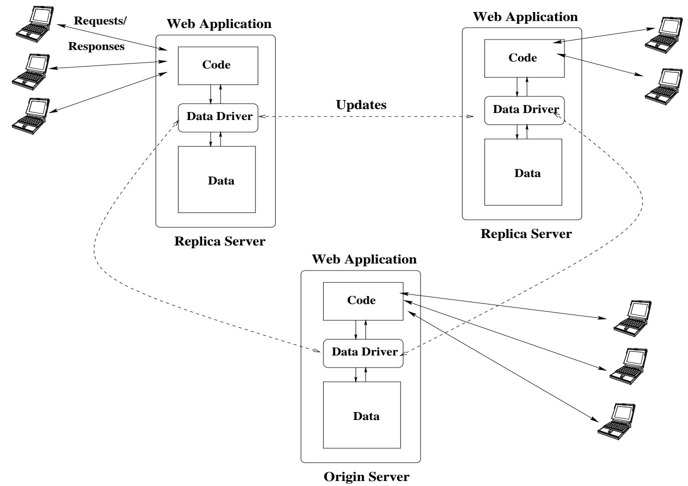

Figure 2: System Architecture

2. APPLICATION AND SYSTEM MODEL

2.1 Application Model

The first and foremost constraint considered in choosing our application model is to keep it *simple* for the application developer. Replication and consistency maintenance that is performed at the application level can possibly achieve the best performance possible. However, this requires the application developer to understand the replication and consistency semantics of the application, which complicates the development process. In our system, we expect the developer to develop business logic code as done for standard EJBs or CGIs, oblivious of the underlying replication. The system will perform data replication and consistency maintenance, transparent to the application code.

The basic application model of our system is given in Figure 1. As seen in the figure, an application consists of code and data. Code is written using standard technologies, such as Active Server Pages (ASPs), CGI scripts and EJBs deployed in an application server. The code receives invocation requests from Web clients, accesses relevant data and generates a result to be returned to the client. The interaction between the code and the data is handled by means of a data driver. We discuss the architecture of the data driver in Section 2.3.

We assume that the data is split into n units, D_1,D_2,...,D_n. Examples of data units are files, database tables, and database records. Each unit is assumed to have a unique identifier, which is used to track it. This identifier can be simply a file name or database table name (provided they are unique). However, for fine-grained data units, such as database records, we assume that the data units can be indexed based on one of their attribute values. This would enable the data driver to locate the data required by the code. Unfortunately, this also limits the system to support only queries that can be translated into indices-based access. It remains to be seen how restrictive this assumption actually is. For example, in the TPC-W benchmark [22], which represents a typical e-commerce application, more than 90% of queries can be answered by indexed data. For data that require non-indexable access, we choose the granularity of the data unit to be higher.

Since data units are the granularity of placement, choosing the correct granularity for data units has important performance implications. If the granularity is too coarse, we may lose the benefit of partial replication as the constituents of the same data unit might have different access patterns. On the other hand, if data units are too fine grained, then it may become difficult to track them individually. In our system, we employ an approach where data units are chosen to be fine-grained and the system clusters the data units with similar access patterns. The system handles replication at the cluster level, thereby making the problem of tracking partial replicas tractable without losing the advantages of partial replication (as the constituents of clusters have similar access pattern). We discuss clustering techniques in Section 3.

2.2 System Model

A system to support partial replication is presented in Figure 2. Each application is hosted on a number of replica servers. Each replica server has a full copy of the application code and a partial copy of the data. Clients are automatically redirected to their closest replica server that hosts the application using standard technologies such as DNS-based redirection.

In our system, updates may originate from multiple sites for the same data cluster. In general, serializing concurrent updates is preferred as this leads to sequential consistency, which is a well-understood model and is also transparent to the application. This is in line with our primary design constraint of keeping the application development process simple. Hence, as a starting point, we plan to limit ourselves to replication strategies that guarantee sequential consistency. We adopt the master-slave replica model as our concurrency control model.

One of the replica servers is chosen to act as the origin server for an application. In addition to hosting a replica, it also decides how data should be clustered and where to place these clusters. The origin server also selects the consistency mechanism to be adopted by each replica.

2.3 Data Driver

The data driver acts as the interface between the code and the data. The data driver preserves distribution transparency of the data, as it hides (from the code) the fact that the data is partially replicated. The data driver has a simple file-system like interface (in case of data stored in files) or JDBC-like interface (in case of data stored in databases). Code invokes these interfaces for accessing the data. A detailed architecture of the data driver is given in Figure 3.

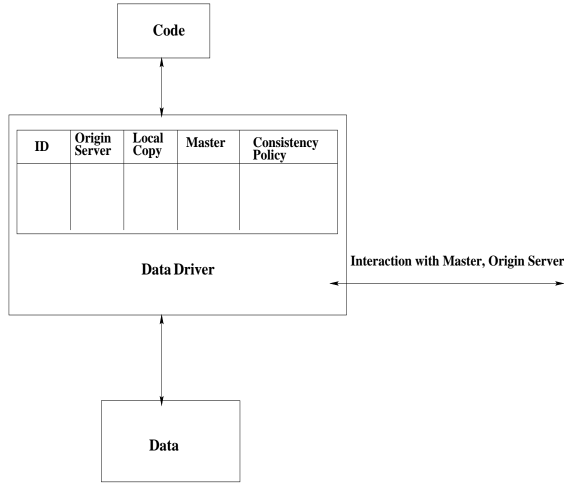

Figure 3: Data Driver Design

The driver is responsible for locating the data unit requested by the code. It does so by first identifying the cluster to which the data unit belongs and then uses the cluster table to find more information about the location of cluster and its policies. This table contains (i) the origin server of the cluster, (ii) a pointer to the cluster in the local system (if available), (iii) its master replica and (iv) the consistency policy information. If the request is for reading a data unit present in a cluster, then the driver locates the cluster using the table and fetches it locally or from the origin server. If the request updates a data unit, the driver updates the data based on the consistency strategy assigned by the origin server. For example, if the replica is configured as a cache, the driver forwards the update request to the master replica.

The details regarding the algorithms used for data clustering, replication of data units and mechanisms used for consistency maintenance are discussed in the subsequent sections.

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. Upon clustering, we need a scalable way to represent cluster membership, i.e., to identify if a data unit belongs to a cluster. In this section, we discuss the issues of data clustering and cluster representation in detail, and present solutions to solve them.

3.1 Data Clustering

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 [13]. 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. The only major difference between the two systems is the scale. Similarity computation in recommender systems typically has ratings of millions of users for millions of items. In our case, we expect thousands of servers to have a access patterns for millions of data units.

We consider two similarity metrics, cosine and correlation-based similarity metric, which are popular in recommender systems.

In cosine-based similarity, the access patterns of data units are treated as m-dimensional vectors and similarity between two data units D_i and D_j is given by

$$Sim(i,j) = cos(A_i, A_j) = \frac{\overrightarrow{A_i}.\overrightarrow{A_j}}{\|\overrightarrow{A_i}\|\|\overrightarrow{A_i}\|} \quad (1)$$

Another method is correlation-based similarity, where similarity is computed as

$$Sim(i,j) = \frac{\sum_{k=1}^{m}(r_{k,i} - \bar{r_i})(r_{k,j} - \bar{r_j})}{\sqrt{\sum_{k=1}^{m}(r_{k,i} - \bar{r_i})^2 \sum_{k=1}^{m}(r_{k,j} - \bar{r_j})^2}} \quad (2)$$

In both methods, data units D_i and D_j are clustered if $Sim(i,j) \geq 1 - x$, for some threshold value x, where $0 \leq x \leq 1$. We need further evaluations to determine which method suits us the best.

This process of clustering can be 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 order $O(m+n)$.

Another important step is to handle the creation of new data units. In our system, creation of a new data unit is done at its origin server. Initially, the data unit is not replicated and all replica servers access it from the origin server. The origin server collects the access pattern for the new data unit and determines its cluster by computing the similarity with the access vectors of cluster. This process requires only $\|C\|$ computations, where C is the total number of clusters. If no suitable cluster is found, the origin server creates a new cluster containing the new data unit. The data unit is replicated as soon as it is inserted into a cluster.

3.2 Cluster Representation

Since the number of data units stored in a cluster is potentially high, we need a scalable scheme to represent cluster membership. A naive scheme would be to use a cluster-to-data- unit membership table that lists the data units contained in each cluster. However, such a scheme will not scale with large number of clusters or data units.

For this reason, we propose to use Bloom filters to represent the membership of items in a cluster [4]. Bloom filters were invented for database systems and have been used in the context of Web systems for the Common Knowledge server [17] and Summary caches [10].

A Bloom filter is a method for representing a set $C = \{C_1, C_2, ..., C_n\}$ of n elements to support membership queries. The idea behind Bloom filters is to allocate a vector v of M bits, initially all set to 0, and then choose k independent hash functions. For each element $c \in C$, the bits at position $h_1(c)$,...,$h_k(c)$ are set to 1 (a bit can be set to 1 multiple times). For a membership query for c, we check the bits $h_1(c)$,...,$h_k(c)$; if any of them is set to 0, then c is certainly not in C. Otherwise, we guess that c is in C, even though it can be a case of false positive. The parameters k and M must be

chosen such that the probability of a false positive is acceptable.

It is impossible to remove an element from a basic Bloom filter, since the bits in a vector can be set multiple times. This pitfall can be removed by replacing each bit with a counter. For addition (deletion) of a new member c, the counter values corresponding to $h_1(c)$,...,$h_k(c)$ are incremented (decremented).

In our system, we use one Bloom filter to represent the membership of each cluster. Filters are created by the origin server and distributed to all replica servers. In addition to a cluster membership table, the origin server also creates a cluster-to-replica- server table, which details the list of replica servers that stores a cluster (using Bloom filters or just a table, depending on the scale).

When a replica server R receives an access request for a data unit D_i, it first checks if D_i is present locally. Otherwise, the replica server finds the list of servers that have the cluster containing D_i (from the server-cluster table). It then forwards the request to the closest one. In case of a false positive, the request is forwarded to the origin server.

4. REPLICATION

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 primarily discuss algorithms concerning data replication and focus on three issues: *replica placement*, *consistency mechanism selection* and *master selection*.

The system performance is affected by all the above three issues. For example, the average client write latency to a given cluster is determined by the placement of replicas and the choice of the master. Similarly, the amount of bandwidth consumed in maintaining consistency depends on the selected consistency mechanism, the choice of the master and the placement of replicas. In our system, we employ an approach where selecting the optimal replica placement, consistency mechanisms and master-slave configuration are not treated as separate problems but as a single problem of selecting an optimal *replication strategy*.

A replication strategy consists of a *placement mechanism*, a *consistency mechanism* and a *master selection*. Placement mechanisms dictate the number and location of replicas. Examples of placement mechanisms include placing replicas at x most popular servers (for some value of x), and heuristics as suggested in [15]. Consistency mechanisms define the protocol that is used to enforce consistency among replicas. Examples of consistency mechanisms include pushing updates from the master to each slave, pulling updates, both possibly combined with invalidations [16] or leases [12]. Master selection mechanisms decide the master replica responsible for handling concurrent updates for a data cluster.

Different replication strategies are likely to provide different levels of performance, so a system designer should be careful in selecting a replication strategy. In earlier work, we studied replication of static Web documents. We showed that no single strategy can universally perform optimal for all Web documents [18]. An important gain in performance can be obtained by associating each document with the strategy that suits it best. We also showed that it is necessary to periodically re-evaluate document-to-strategy associations, as changes in documents' access and update patterns are likely to affect system performance [20]. These studies were performed for static documents, whose updates originated from a single server. We expect these results to be also valid in the case of application replication. In view of these earlier findings, we propose to select strategies on a per-cluster basis and re-evaluate selections periodically, or when there is a significant change in its access or update patterns.

The "best" strategy for each cluster will be selected among a set of candidate strategies. This is done by maintaining logs of read and write accesses to the cluster and evaluating the performance that each candidate strategy would have delivered in the recent past with the recorded access patterns.

The performance of each strategy is measured using a *cost function*. This function is designed to capture the inherent tradeoff between performance gain by replication to performance loss by consistency enforcement [18]. An example of cost function that measures performance of a strategy s during a time period t is follows:

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

where r is the read latency, w is the write latency, b is the amount of bandwidth used for consistency enforcement, and α, β and γ are weights associated to each metric. These weights must be set by the system administrator based on the system constraints and application requirements. A bigger weight implies that its associated metric has more influence in selecting the "best" strategy.

In our system, the origin server computes this cost function value for each candidate strategy using trace-driven simulations. The "best" strategy is selected as the one that yields the minimum cost. When necessary, the system will dynamically switch strategies.

The selection of the best strategy using this method can be computationally expensive, as it requires to evaluate every combination of replica placement, consistency mechanism and choice of master. If the computational overhead turns out to be high, we propose to reduce the search space. For example, one can statically select a master using a heuristic such as selecting the server with most number of writes. Then, selecting a replication strategy only requires to explore all combinations of replica placement and consistency mechanisms. This selection method reduces the number of evaluations, thereby reducing the computational overhead. Further work is needed to determine the effectiveness of this method.

5. RELATED WORK

For the past decade, numerous solutions have been proposed in the context of caching systems for delivering Web content [21]. Most systems assume that the temporal locality of the client requests is high, as these systems were initially built for delivering static Web documents. Unfortunately, this assumption is often wrong for dynamic applications. To handle Web applications, some CDNs employ an approach where only static fragments of the dynamic documents are cached and complete documents are re-assembled by the caches for each request [2, 1]. However, these systems are not suitable for hosting applications characterized by a large number of unique reads or a significant number of writes. Such applications can be distributed only using replication, where the application code is executed at the replica servers.

Many systems exist that perform code distribution [3, 6]. However, these systems do not perform data replication and are suited for applications that require small amount of data transfer and spend most of their time in local computations.

In [19], the authors present a CDN for application replication, where the code is replicated along with the data. However, the CDN is limited to applications, where data are updated only by the origin server.

A similar CDN for application replication is built by Akamai and IBM, using Edge Computing Infrastructure [14] and WebSphere [9]. In this system, the code is replicated at the edge servers. The data are also stored in the local replica server using a Java embedded database called Cloudscape [8]. However, in this system, the database is configured as write-through cache, where each write is

forwarded to the origin server. This may increase the write latency as each write will incur wide-area network latency.

In [11] 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 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.

6. CONCLUSION

This paper explores the design space of systems that perform application replication. We adopt a simple application model for the system, which we hope will ease the process of application development. The novelty of our approach is that it employs partial replication by segmenting the application data into data units, grouping units with similar access patterns into a cluster and replicating clusters independently from each other. We believe partial replication will reduce the consistency overhead as updates are sent only to replica servers that access them. This allows the system to exploit location-specific interests in request patterns. Key issues that must be addressed to build this system are *data clustering*, *cluster representation* and *cluster replication*. We discussed the research problem contained in these issues and suggested solutions to solve them.

We are currently working on building a prototype and plan to test it with the TPC-W benchmark. We are also planning to investigate and evaluate different heuristics for selection of replication strategies.

7. REFERENCES

[1] *Akamai Edgesuite Architecture,* `http://www.akamai.com/en/html/services/edge_proc_targeting.html`.

[2] *ASP .NET Caching Features,* `http://authors.aspalliance.com/aspxtreme/webapps/aspcachingfeatures.aspx`.

[3] A. Awadallah and M. Rosenblum, *The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution*, 7th Web Caching Workshop, August 2002.

[4] B. H. Bloom, *Space/time tradeoffs in hash coding with allowable errors*, CACM (1970), 422–426.

[5] E. Brewer, *Lessons from giant-scale services*.

[6] P. Cao, J. Zhang, and K. Beach, *Active Cache: Caching Dynamic Contents on the Web*, Middleware '98 (Berlin), IFIP, Springer-Verlag, September 1998, pp. 373–388.

[7] V. Cate, *Alex – A Global File System*, File Systems Workshop (Berkeley, CA), USENIX, USENIX, May 1992, pp. 1–11.

[8] IBM Corporation, *IBM Cloudscape*.

[9] IBM Corporation, *IBM Websphere*.

[10] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder, *Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol*, IEEE/ACM Transactions on Networking **8** (2000), no. 3, 281–293.

[11] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar, *Application specific data replication for edge services*, 2003, pp. 449–460.

[12] Cary Gray and David Cheriton, *Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency*, 12th Symposium on Operating System Principles (New York, NY), ACM, ACM Press, December 1989, pp. 202–210.

[13] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl, *An algorithmic framework for performing collaborative filtering*, Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, ACM Press, 1999, pp. 230–237.

[14] Akamai Inc., *Akamai Edge Computing Architecture*.

[15] J. Kangasharju, James Roberts, and K.W. Ross, *Object Replication Strategies in Content Distribution Networks*, 6th Web Caching Workshop (Amsterdam), North-Holland, June 2001.

[16] B. Krishnamurthy and C. Wills, *Piggyback Server Invalidation for Proxy Cache Coherency*, Computer Networks and ISDN Systems **30** (1998), 185–193.

[17] Hannes Marais and Krishna Bharat, *Supporting cooperative and personal surfing with a desktop assistant*, ACM Symposium on User Interface Software and Technology, 1997, pp. 129–138.

[18] Guillaume Pierre, Maarten van Steen, and Andrew S. Tanenbaum, *Dynamically selecting optimal distribution strategies for Web documents*, IEEE Transactions on Computers **51** (2002), no. 6, 637–651.

[19] Michael Rabinovich, Zhen Xiao, and Amit Agarwal, *Computing on the edge: A platform for replicating internet applications*, Proceedings of the Eighth International Workshop on Web Content Caching and Distribution (WCW'03) (Hawthorne, NY, USA), September 2003.

[20] Swaminathan Sivasubramanian, Guillaume Pierre, and Maarten van Steen, *A case for dynamic selection of replication and caching strategies*, Proceedings of the Eighth International Workshop on Web Content Caching and Distribution (WCW'03) (Hawthorne, NY, USA), September 2003.

[21] Swaminathan Sivasubramanian, Michal Szymaniak, Guillaume Pierre, and Maarten van Steen, *Web replica hosting systems*, Tech. Report IR-CS-001, Vrije Universiteit, Amsterdam, The Netherlands, May 2003.

[22] Wayne Smith, *TPC-W: Benchmarking an e-commerce solution*.

[23] Haifeng Yu and Amin Vahdat, *Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services*, ACM Transactions on Computer Systems **20** (2002), no. 3, 239–282.