Replicating Web Applications On-Demand

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

Abstract

Many Web-based commercial services deliver their content using Web applications that generate pages dynamically based on user profiles, request parameters etc. The workload of these applications are often characterized by a large number of unique requests and a significant fraction of data updates. Hosting these applications drives the need for systems that replicates both the application code and its underlying data. We propose the design of such a system that is based on on-demand 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 proposed system allows complete replication transparency to the application, thereby allowing developers to build applications unaware of the underlying data replication. We show that the proposed techniques can reduce the client response time by a factor of 5 *in comparison to existing techniques for a real-world e-commerce application used in the TPC-W benchmark. Furthermore, we evaluate our strategies for a wide range of workloads and show that on-demand replication performs better than centralized and fully replicated systems by reducing the average latency of read/write data accesses as well as the amount of bandwidth utilized to maintain data consistency.*

1. Introduction

The Web is the leading platform for hosting commercial services, such as book shops and music stores, in the Internet. The Web sites hosting such services often do not deliver static Web pages, but are made of applications that generate pages based on request parameters, individual user profiles, etc. Dynamic document generation for a request usually requires, in turn, to issue read or write accesses to a database.

Hosting such applications in a centralized server (or cluster of servers) may result in poor response time for Web clients due to the wide-area network latency introduced for each request. An obvious solution, known as fragment caching, is to cache the pages generated by the applications at servers located close to the clients [1, 11]. However, this solution relies on the assumptions that the temporal locality of requests is high and the requests that lead to data updates are infrequent. Unfortunately, this assumption is often not valid for applications that receive a large number of unique requests or a significant number of requests that lead to data updates. Such applications can be distributed only through replication, where the application code is executed at the replica servers. This avoids the wide-area network latency for each request 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 can reduce the latency of requests, as the requests can be answered by the application hosted by the server located close to the clients. Replicating applications is relatively easy provided that the code does not modify the data [14]. However, most applications do modify their underlying data. In this case, it becomes necessary to manage data consistency across all replicas.

Propagating every data update to all servers can lead to a significant overhead in terms of update traffic if the application generates many data updates. Such overhead can be reduced by adopting weak consistency models, but this requires significant expertise from the application developers. In our system, we made the opposite choice and focus on scalable solutions that guarantee full replication transparency for Web applications while maintaining strong consistency.

In this paper, we propose to not replicate all application data at all servers. Instead, data are segmented into data units and each data unit is replicated only to the servers that access it frequently. We call this approach *on-demand replication*. This approach can reduce the synchronization overhead as consistency updates for a given data unit must be sent to a reduced number of servers. Furthermore, we propose to combine our technique with fragment caching. As we later show in our performance studies, this combination can perform well for a wide range of application workloads.

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 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. As we show later in our performance evaluations, on-demand data replication can reduce the update traffic by 1 to 2 orders of magnitude in comparison with other existing techniques. In addition, for the TPC-W e-commerce benchmark [16], on-demand replication can reduce the client latency by a factor of 5 in comparison with other existing techniques.

The contributions of this paper are as follows: (i) we propose an architecture for a system that performs on-demand replication of Web applications; (ii) we show that such on-demand replication can provide significant performance gains using a real world e-commerce application and its workload provided by the TPC-W benchmark; and (iii) we evaluate the potential performance gains of on-demand replication for a wide range of workloads, characterized by different update ratios and data access patterns.

The rest of the paper is organized as follows: Section 2 presents our application and system model. Section 3 presents the detailed design of the data driver, the central component of our on-demand data replication architecture. Section 4 discusses our replication and caching techniques. Section 5 evaluates the performance gains due to on-demand replication for the TPC-W based e-commerce application. Section 6 evaluates the performance gains due to on-demand replication for a wide range of application workloads. Section 7 discusses the related work and Section 8 concludes the paper.

2. System Model

2.1. Application Model

An important issue when replicating an application is to decide to which extent the application code 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.

We made the opposite design choice by having a completely *replication-transparent* application model. In our system, the application developer need not worry about replication issues but only sticks to functional issues. The system will automatically derive a replication strategy, and possibly adapt it under changing access patterns.

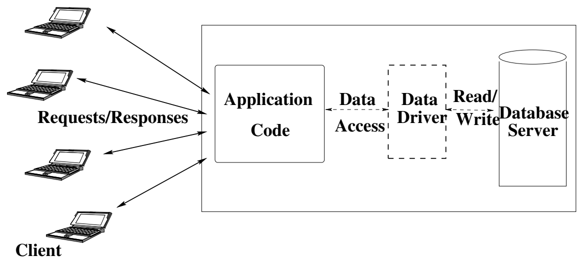


Figure 1. Application Model

To keep replication transparent to the application developers, we decided that our system should provide sequential consistency [12]. This consistency model enables the developer to write applications as if the underlying data were concurrently accessed from a centralized location, thereby ignoring distribution issues.

Our application model is shown 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 to generate a response.

Access to the data is realized by a data driver, which acts as the interface between the code and data.[1] The data driver preserves distribution transparency of the data as it hides the fact that data are partially replicated. It has a simple JDBC-like interface and 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 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 database tables, or even records. The system replicates each data unit according to its specific pattern.

Choosing the right data granularity 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 may 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,

1 The data driver we describe is different from conventional JDBC drivers as this driver is not just an interface driver but also responsible for functional aspects such as replication and location of data.

thereby making the problem of tracking a cluster tractable without losing the benefits of partial replication.

In [8], the authors study a similar problem of clustering Web pages to reduce the overhead in handling replicas for each Web page. The authors propose spatial clustering algorithms to group Web pages into clusters. Subsequently, the system replicates pages at the cluster level, thereby reducing the cost of replica placement. The authors also propose incremental clustering algorithms to handle changing access patterns and creation of new Web pages. The authors show that these clustering algorithms perform well for real-world Web traces. We plan to use similar spatial clustering algorithms for clustering data units. However, data clustering is not the focus of this paper hence, throughout this paper, we assume the data units are already clustered.

2.2. System Architecture

The architecture of our proposed system is presented in Figure 2. A given application is hosted over m edge servers spread across the Internet. Each client is assumed to be redirected to its closest edge server using standard technologies, such as DNS-based redirection [1, 9]. Communication between edge servers usually goes through wide-area networks incurring wide-area latency.

When a client issue an HTTP request to the Web server, the server first checks if the response to the corresponding request is present in the cache. If found, the response is returned immediately from the cache. Otherwise, the request is passed over to the application code residing in the application server. The application code usually issues a number of read/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 requested data are not present locally. Additionally, when handling write data accesses, the driver is also responsible for ensuring consistency with other replicas of the updated data unit.

As noted earlier, we want to maintain sequential consistency. We adopted a simple master-slave consistency protocol: each data cluster has a master server responsible for serializing concurrent updates emerging from different replicas. Read data accesses are forwarded to the closest server that contains a replica. Write accesses are always forwarded to the master of the data cluster, which immediately pushes the update to all its replicas. Issuing all write operations to a given cluster at a single location effectively serializes updates, which generates sequential consistency.

To perform on-demand replication, the system must cluster data units, decide on the placement of replicas for each 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 clustering and placement periodically to handle changes in data access patterns and the creation of new data units.

3. Data Driver

The data driver is the central component of our system. It is in charge of clustering data units, replicating them, locating the data units required by the application code and maintaining consistency among replicated data.

The driver maintains two tables. First, the *cluster-membership* table stores the identifiers of data units contained in each cluster. [2] Second, the *cluster-property* table contains the following information for each data cluster: the origin server of the cluster, a reference to the cluster in the local database (if available), the identifier of its master replica and the list of servers that host a copy of this cluster. These two tables are fully replicated at all edge servers.

The driver locates a data unit by first 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.

The data driver receives two kinds of data access queries. We refer to the queries based on primary keys of a table as *simple queries*. Example of a simple query can be "Find customer record whose userid is 'Bob'." Queries based on non-primary keys (e.g., secondary key based access) are referred to as *complex queries*. Example of a complex query can be "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 onto required data units, which makes the processing of locating data unit(s) relatively straightforward.

For answering complex queries, the driver must not only check its local database but also the entire database located across multiple servers, as it does not have the complete information on which data units match the query (e.g., there can be data units on other servers whose location field value is "Amsterdam"). To do this, the driver need not contact all servers but only a subset of servers that in total have the complete database table. We call this subset a *min-set*. The driver then requests the data drivers in this set of servers to execute the query, merges their responses and returns the result to the application code.

2 A naive representation of this table can lead to a scalability bottleneck. To avoid this, we intend to use Bloom filters [3].

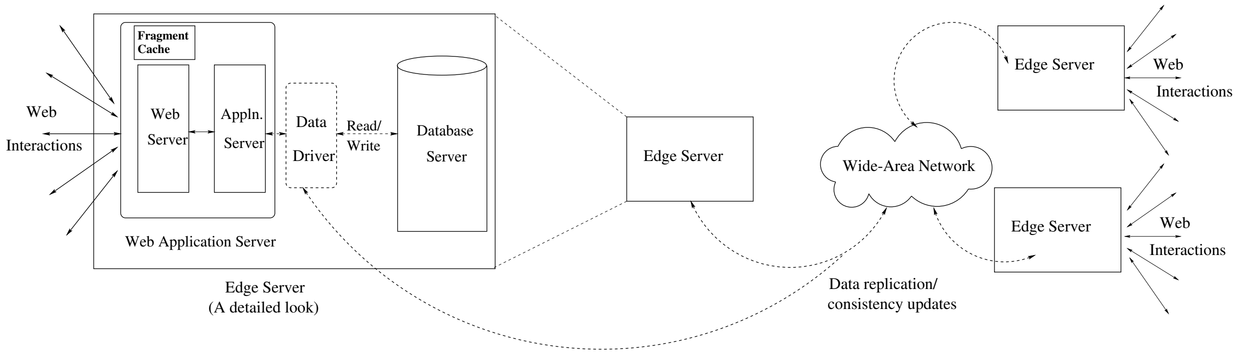


Figure 2. System Architecture - Edge servers serving clients close to them and interactions among edge servers goes through Wide-area network. A detailed look of the design of an Edge Server.

Note that there can be more than one *min-set* and selecting the right one is important for having a low response time. Since queries can be addressed to the *min-set* in parallel, the response time for answering a complex query is limited by the maximum round-trip time to any server in the *min-set*. The problem of finding a *min-set* for a given database table is to determine a set of servers that (i) together have the entire database table and (ii) offer the smallest round-trip time from the requesting server. To explain this, let us consider the scenario where the system has 3 servers $\{R_1,R_2,R_3\}$ and 4 data clusters in total. Let us assume that R_1 contains cluster $\{C_1,C_2\}$, R_2 contains $\{C_3,C_4\}$ and R_3 contains $\{C_1,C_3\}$. If server R_1 gets a complex query, then the *min-set* for R_1 will be $\{R_1,R_2\}$, if R_2 has smaller round-trip time to R_1 than R_3. Determining a good *min-set* for a server is relatively simple. Further, it needs to be re-computed only when there is a change in the *cluster-property* table, i.e., when an edge server is added/removed from list of servers holding the replica of a cluster.

Note that, although querying a *min-set* incurs wide-area communication, the proposed system does not perform any worse than existing systems while answering these complex queries. For instance, in fragment caching systems, answering both simple and complex queries involves wide-area traffic. To obtain an idea of the percentage of transactions that involve simple and complex queries in a real-world application, we examined the TPC-W benchmark application and its different workload mixes [16]. Depending on the workload mixes, at least 60% of the transactions are based on simple queries, while the rest use complex queries. This means that wide-area communication overhead occurs in at most 40% of the cases. As we show later, this overhead can be reduced by fragment caching.

4. Replication and Caching 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 our system, each data cluster is replicated independently. In this section, we discuss algorithms concerning the selection of the "best" replication strategy for a data cluster. Further, we discuss fragment caching techniques which are useful for application with mostly read-only data accesses and propose a hybrid strategy that is made of a combination of on-demand data replication and fragment caching.

4.1. Data Replication

A replication strategy describes three aspects: *replica placement*, *consistency mechanism*, and, in our case, *master selection*. Replica placement decisions concerns the number and location of replicas. Consistency mechanism dictates the protocol used to enforce data consistency among replicas. Master selection involves choosing a master replica that is 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 data driver 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. But 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.

In general, there is a clear tradeoff between the performance gain due to replication and the performance loss due to 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 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.

4.1.1. Replica Placement Heuristics In our system, the origin server periodically collects the access patterns of each data cluster from all edge servers. Subsequently, it places a replica of a data cluster in a 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. On the other hand, a low value of x will lead to a fully replicated configuration. 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 the right value of x is not reasonable, as the number of parameters that affect the system performance is high. Instead, in our system, administrators are just expected to define their preferred performance tradeoffs by choosing the weight parameters of the cost function. The origin server will automatically adjust the replication configuration to the one that gives the lowest cost.

4.1.2. Master Selection Heuristics 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. The *most-writer* heuristic selects the master as the replica server that generates the highest number of write accesses. This strategy allows the highest fraction of update accesses to be handled locally. However, if all servers issue similar numbers of update accesses, this strategy may give poor write latency because a large fraction of update access 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. Let n_i be the number of write access requests received by replica server R_i and l_{ij} be the latency between replica server i and j (we assume that latency measurements between servers are symmetric, i.e., l_{ij}=l_{ji}). 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.

4.2. Fragment Caching

Data replication is required when an application's workload is characterized by a large number of unique requests and/or significant number of writes. However, if the system exhibits some temporal locality among requests, then caching techniques are shown to be useful [4].

One of the most widely used techniques for caching results of Web applications is fragment caching [7]. The idea behind this technique is to cache the responses for popular requests in the Web server. This avoids the overhead in regenerating the response for the popular requests. This is suitable for requests that do not modify application data and are not unique (e.g., request for local weather).

In fragment caching, the data driver instructs the application server to cache a query response, if the write-to-read request ratio to the underlying data units that were accessed is below a threshold, $Cache_{max}$. To ensure the consistency of responses, if a fragment response is cached, the data driver maintains a *dependency graph* of the fragment responses with a particular data unit. An example of this graph is the dependency between a data unit that contains the stock price of a company and the fragment response that lists the stock price. Later, if the data unit corresponding to the stock price is updated, then the relevant fragment is invalidated, thus preventing the Web server from delivering any stale fragment. Similar consistency mechanisms have been shown to be scalable for Web pages [5, 7].

4.3. Hybrid strategy

Many applications are characterized by a workload where a sizeable fraction of requests are cacheable, while the rest are unique and contain significant number of writes. For example, in the TPC-W benchmark, which represents an online bookstore application, 5-10% of the requests are for the new-products page which lists the latest books. Since updates to the new-products page occur rarely (0.11% of requests), the fragment responses to this page can be cached. Further, at least 50% of the benchmark's requests are unique (e.g., pages related to customer profile) with a considerable fraction of them leading to data updates (e.g., those for updating shopping carts or for ordering books). Such scenarios warrant data replication to ensure fast response time with minimum consistency overhead.

In our system, we propose to use fragment caching and on-demand data replication simultaneously, where the fragment cache is located in the Web server and the on-demand data replication is performed by the data driver. We call this hybrid technique as ODRC (On-demand replication with caching). This strategy can reduce the wide-area communication overhead involved in generating pages with complex queries by caching these pages, provided the updates to the underlying data occur less frequently.

5. TPC-W Benchmark: A Test Case

To study the impact of on-demand replication on real-world applications, we evaluated the performance of our system using a well-known e-commerce transactional benchmark, *TPC-W*. In this section, we present an overview of TPC-W, describe our simulation setup and discuss our results.

5.1. Overview

TPC-W benchmark is an industry standard transactional benchmark that models an on-line bookstore, where the customers search, shop and order books [16]. It is aimed to represent a typical e-business Web site.

The TPC-W application stores its data in 7 tables: Book, Customer, Address, Order, Cart, Orderline and Cartline. It handles 14 kinds of customer requests (also known as Web Interactions): Home page, New Products page, Best Sellers, Product Detail, Search Request, Search Results, Shopping Cart, Customer Registration, Buy Request, Buy Confirm, Order Inquiry, Order Display, Admin Request, and Admin Confirm. Among these interactions, the first six are browsing interactions, which lead to read-only data accesses. The rest are ordering-related interactions and mostly lead to data updates. A typical user shopping pattern, referred to as a user session, comprises of number of these Web interactions. The transition probabilities for switching between different Web interactions in a user session is dictated by the the benchmark's workload mix.

The benchmark defines three workload mixes: browsing, shopping, and ordering. The browsing mix consists of 95% browsing interactions and 5% ordering interactions. The ordering mix consists of 50% ordering interactions and 50% browsing interactions. The shopping mix consists of 80% browsing interactions and 20% ordering interactions.

The performance of the TPC-W benchmark is measured primarily by two metrics: (i) WIPS (number of Web Interactions Per Second), which measures system throughput, and (ii) WIRT (Web Interaction Response Time), which measures system efficiency. In our experiments, we are more interested in the efficiency of the system (i.e., WIRT) than its throughput (WIPS). This is because the system throughput can be increased by deploying more hardware resources, for instance by deploying a server farm instead of a single server. We believe that this does not necessarily result in better WIRT, as the inevitable overhead in Web application replication is not the server's throughput performance but the network latency incurred for each transaction.

5.2. Simulation setup

Building a fair experiment to simulate an Internet-wide system hosting a Web application involves two important challenges: simulating a wide-area network with realistic network delays between the edge servers placed worldwide; and simulating the diversity of interest among clients in each particular piece of data.

To simulate a set of servers to host our application, we selected 100 hosts that visited our department Web site, such that they were spread across 6 continents and 66 countries, and could represent our edge servers. We estimated the latencies between each pair of hosts using SCOLE [17]. 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. It must be noted that the potential inaccuracies of latency estimations are not a problem for our experiments, as we are only interested in a realistic set of latencies rather than precise latency measurements between the actual servers.

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 interests influences the performance of on-demand replication. For example, if a data unit 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 unit is of equal interest to clients of all replica servers, then the performance gains due to on-demand replication will not be lower. However, it is important for us to study the performance of on-demand replication in the full spectrum of cases.

Unfortunately, TPC-W does not specify any standard pattern regarding the diversity of client interest. In our experiments, we modelled the diversity of client interests 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.[3]

We clustered the books into 100 non overlapping *book-clusters*. The geographical preference of a server to a book-cluster is modelled as follows. For a book-cluster j, server i is assigned a rank, $r_{ij} = i - j$, if $i \geq j$, and $100 - i - j$, otherwise. Using these rank values, the frequency of occurrence of request from server i for a book-cluster j, is generated: $f_{ij} = C \cdot r_{ij}^{-a}$. Then, these frequency values are normalized so that we can derive a series of load values L_{ij} whose sum is 1 (i.e., $L_{ij} = \sum_{i=1}^{m} f_{ij}/m$, where m is the number of edge servers). Subsequently, a client request for a book cluster j is assigned to a server i with the probability L_{ij}. Within a book-cluster, books are selected with equal probability, as this does not affect the Zipf distribution of servers to the book-clusters.

To study the system using diverse access patterns, we varied the parameter a. A trace generated with a low value of a implies that each server is equally likely to access a book-cluster i.e., the distribution of client interests is flat. In contrast, a high value of a generates a trace where each book-cluster is of interest only to a small number of servers (those with the lowest rank values) i.e., the distribution of client interest is more skewed. In our simulations, we varied a between 0 and 3. We did not vary interest parameters for other database tables as they are usually not accessed by requests from multiple users (e.g., customer records table, shopping cart table).

While evaluating the WIRT, we just took into account the wide-area network latency incurred by a request. This is because we assume that the latency between client to its local edge server and the request processing overhead in an edge server are negligible compared to the latency incurred in communicating with servers that must be reached through a wide-area communication.

In our experiments, we study the performance of four different system configurations: (i) Centralized, (ii) Fragment Caching (FC), (iii) On-Demand Replication (ODR) and (iv) ODR with Caching (ODRC). The first system, Centralized, has the application server and database server at the origin server and hence each request incurs wide-area network traffic. To be fair on the centralized solution, we chose the best possible replica server as the origin server, i.e., the server with the minimum average latency to other servers is chosen as the origin server. The second system, FC, has edge servers around the world and uses the fragment cache technique with the threshold $Cache_{max}$ fixed at 1% (see Section 4.2). Edge server caches are assumed to have infinite storage capacity[4]. The third system, ODR, simulates our proposed approach of on-demand data replication with closest-writer master selection and $P5$ placement heuristic. The ODRC system uses the hybrid technique described in Section 4.3 with closest-writer master selection and $P5$ placement heuristic.

We studied the WIRT of these 4 systems for 10^7 user sessions, divided equally among the 100 edge servers. Each user session consists of at most 10 Web interactions and all interactions are handled by the same server. The time between the interactions in a session was varied between 2 to 8 seconds. The simulations were warmed up and the results of the first 10^5 sessions were discarded. The simulations were run repeatedly to gain a confidence interval of 95%.

5.3. Results

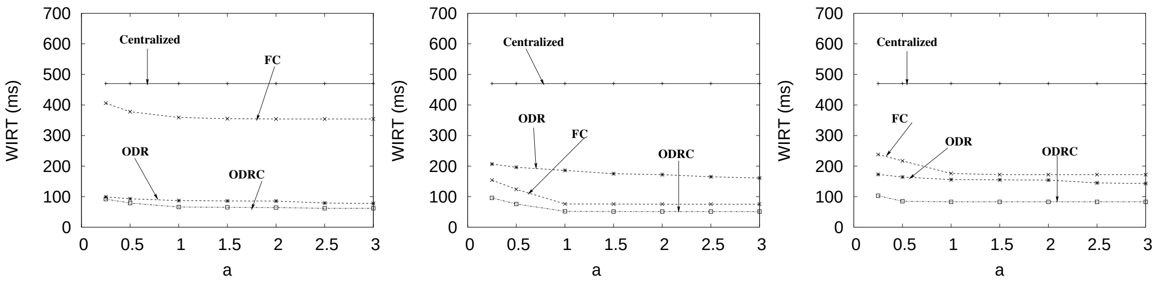
Figure 3 shows, for each workload mix, the WIRT of all systems for different values of a. As seen in the figure, ODRC performs the best for all workload mixes while Centralized performs the worst.

For the ordering mix, which has the highest fraction of ordering interactions, ODR and ODRC outperform FC and centralized systems by a factor of 5. This clearly illustrates the advantage of on-demand data replication, as it enables the system to perform local updates without incurring wide-area traffic. Note that the gain in WIRT by ODRC in comparison to ODR is low. This is because this workload mix has a very small fraction of popular cacheable requests and hence the gain due to caching is not significant.

For the browsing mix, which consists of 95% browsing and 5% shopping interactions, the caching systems perform best. Among the caching systems, ODRC performs better as it performs not only caching of responses but also data replication, though the gain in WIRT in comparison to FC is only by a margin of 50%. This is due to the fact that requests are mostly read-only and caching suffices for such scenarios. Among the rest, FC outperforms its ODR counterpart, as ODR does not capture the temporal locality of requests (exhibited for high values of a). ODR also incurs

3 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 occurrence.

4 It must be noted that an infinite cache size is not a realistic assumption. However, this was done to compare our proposed system with the best possible caching configuration.



(a) Ordering Mix (b) Browsing Mix (c) Shopping Mix

Figure 3. Effect of a on WIRT for different TPC-W workload mixes

wide-area network traffic to generate responses to interactions involving complex queries, such as Best Sellers and New Products interactions (which constitutes 22% of the workload). This overhead is avoided by ODRC as it caches the responses to these interactions.

For the shopping mix, which constitutes 80% browsing and 20% ordering interactions, ODRC outperforms all its counterparts by at least 200%. Among the rest, the gain in WIRT by ODR in comparison to FC is not very high. This is because the gain of performing local updates in ODR is compensated by the penalty incurred for computing the complex query for interactions, such as Best sellers and New Products, each time. While the FC avoids the latter problem by caching the responses, it incurs wide-area latency for all update requests. ODRC enjoys the benefits of these two systems, as ODR and FC optimize different subsets of requests, thereby leading to a significant improvement in WIRT.

It can be seen from the figures that the Zipf variable a does not have a major influence on the WIRT. This is due to the fact that the requests to individual book records constitute at most 20% for browsing mixes and even less for other workloads. For higher values of a, caching systems perform well as it implies high temporal locality among requests. Even then, the gain in WIRT is low because book detail interactions constitutes only at most 20% of the average WIRT.

6. Performance Evaluation of Replication Strategies

6.1. Simulation Setup

In the previous section, we studied the performance gains obtained due to on-demand replication techniques for a well known e-commerce application by varying the access pattern for the book records. However, that study was quite restrictive as the update ratio to the book records is at most 0.11%, with requests to these records constituting no more than 30% of the total workload.

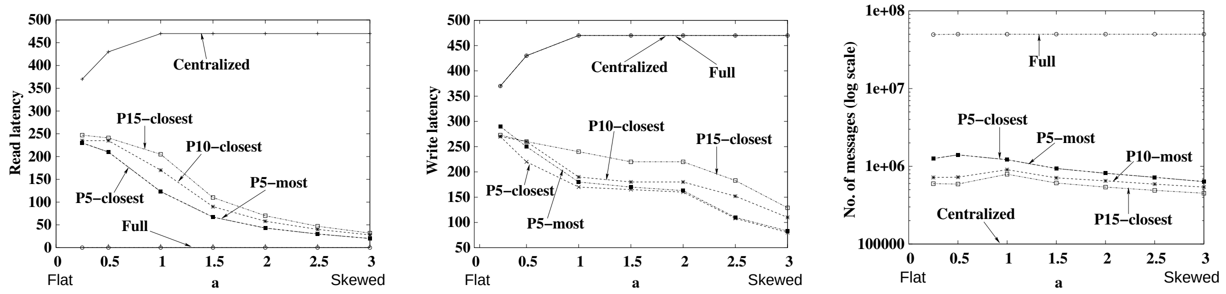
In this section, we examine the performance of our techniques for a wide range of workloads with full spectrum of write ratios and geographical interest distributions. To this end, we simulated a data driver which receives read and write access requests to its data units. In this section, we limit our study to replication strategies only as fragment caching does not influence the performance of the data driver.

We selected the origin server and 100 edge servers as in the previous experiment. Similar to the previous experiment, we varied the interest pattern of clients using Zipf distributions. We measured 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.

6.2. Influence of a on performance

In our first set of experiments, we study the effect of client interest pattern on system performance. We fixed the write ratio at 0.5. Each simulation consists of $1,000,000$ requests. We study the performance of the following strategies: (i) centralized solution, (ii) fully replicated solution with origin server as the master (Full), (iii) $P5 - closest$ (ODR technique with $P5$ placement and $closest - writer$ master selection heuristic), (iv) $P10 - closest$, (v) $P15 - closest$ and (vi) $P5 - most$. Due to the lack of space, we present only some of our simulation results. For a detailed performance evaluation, please refer to [15].

Figure 4 presents the effect of varying the value of a on the system performance. As can be seen, on-demand replication produces a significant gain in terms of read/write latencies (by a factor of 4) and reduced update traffic (by two



(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 4. Effect of distribution skew on system performance

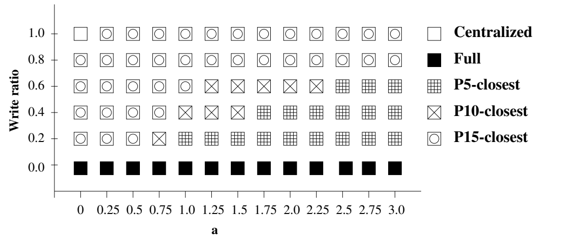


Figure 5. Best Replication Policy for Different Access Patterns

orders of magnitude) compared to fully replicated or centralized systems. The more the client diversity increases, the better 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, as the Internet is often affected by network congestion. A reduced number of consistency messages also will lead to improved write latency of the system and less cost, as content delivery systems are usually charged by Internet Service Providers (ISPs) and data centers based on the amount of traffic they generate.

6.3. Best Performing Strategy

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. For each access pattern, we evaluated the value of the cost function for each replication strategy 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=$1/b_{max}$, where r_{max}, w_{max} and b_{max} are maximum values of average read latency, write latency, and number of consistency messages, respectively. Figure 5 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.

7. Related Work

A number of systems have been developed to handle Web application replication [1, 2, 6, 14]. 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 accesses need to be forwarded to a single remote location irrespective of their update patterns. In contrast, we propose to select the master replica for each data unit based on its individual update pattern, which potentially results in a low write latencies.

In [10], 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 the application's consistency and distribution semantics. 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 [13], a distinction being that in distributed databases, fragments are usually not created based on runtime analysis of access patterns. Using traditional distributed database technologies, partitioning must be done by a clever administrator who has a deep knowledge of the application semantics and its access patterns. However, in our system we propose to do this automatically and dynamically based on the access patterns of the application data. However, further research is needed to substantiate our claim of scalability for real Web applications.

8. Conclusions and Future Work

In this paper, we propose a system for hosting Web applications that performs on-demand data replication. 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 are replicated only to servers that access them often. This allows the system to exploit location-specific interests in request patterns. Furthermore, we also propose a scheme to combine existing caching techniques with on-demand data replication. We showed that our techniques can reduce the client response time by a factor of 5 in comparison to existing techniques for a real-world e-commerce application, such as the TPC-W bookstore. Furthermore, we evaluated our strategies for a wide range of workloads. We showed 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 bandwidth utilized to maintain data consistency. Moreover, we showed that the best replication strategy depends on the data cluster's access pattern and proposed a scheme where our system will automatically select the best replication strategy for a given situation through run-time evaluation of a cost function. We are currently working on the implementation of a data driver that operates on top of the MySQL database.

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] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. On the implications of zipf's law for web caching. In *Proceedings of 3rd International WWW Caching Workshop* (1998).

[5] CAO, P., AND LIU, C. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers 47*, 4 (1998), 445–457.

[6] 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.

[7] CHALLENGER, J., DANTZIG, P., AND WITTING, K. A fragment-based approach for efficiently creating dynamic web content. *To appear in the ACM Transactions on Internet Technology* (2004).

[8] CHEN, Y., QIU, L., CHEN, W., NGUYEN, L., AND KATZ, R. H. Clustering web content for efficient replication. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP'02)* (2002).

[9] FEI, Z., BHATTACHARJEE, S., ZEGURA, E. W., AND AMMAR, M. H. A novel server selection technique for improving the response time of a replicated service. In *INFOCOM (2)* (1998), pp. 783–791.

[10] 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.

[11] 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.

[12] LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers 28(9)*, Sep. 1979, pp. 690-691.

[13] OZSU, P., AND VALDURIEZ, P. Principles of distributed database systems, 2nd edition, Prentice Hall, 1999.

[14] 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).

[15] SIVASUBRAMANIAN, S., PIERRE, G., AND VAN STEEN, M. A system for on-demand Web application replication, Dec. 2004. `http://www.globule.org/`.

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

[17] SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. Scalable cooperative latency estimation. Accepted for publication in ICPADS, Dec. 2004. `http://www.globule.org/`.