



# FRACS : A Hybrid Fragmentation-Based CDN for E-commerce Applications

**FRACS : un CDN Hybride pour les Applications de Commerce Electronique Base sur la Fragmentation**

IKRAM CHABBOUH AND MESAAC MAKPANGOU  
projet REGAL, INRIA Rocquencourt  
janvier, 2007

## Abstract :

In order to accelerate access to Web applications, content providers are increasingly relying on Content Delivery Networks. Currently, CDNs serve the dynamic content from the edge in two major ways : page assembly or edge computing. Page assembly assumes that the proportion of cacheable content is high, that the cached fragments are reusable and that they do not change very often, while edge computing generally assumes that the whole application is replicated on the edge, which is not always suitable. Besides, current CDNs do not provide a means of scaling the database component of a Web application. In this study we propose a hybrid CDN called FRACS which combines both page assembly and edge computing in order to address the needs of relatively static applications as well as more dynamic applications. FRACS automatically determines the replicable fragments of a Web application, then it modifies the application's code so as to generate fragmented pages in ESI format and to enable the server to serve the fragments separately. Moreover, FRACS maintains the consistency of all the manipulated fragments. Using the TPC-W benchmark we were able to achieve up to 60% savings in bandwidth and more than an 80% reduction in response time.

## Key Words :

CDN, Web Applications, Dynamic content, cache and replication.

## Resume :

En vue d'accroître l'accès au contenu dynamique, les fournisseurs de contenu s'adressent de plus en plus aux réseaux de distribution du contenu (CDNs). Actuellement, les CDNs délivrent le contenu dynamique de deux manières différentes : l'assemblage des pages ou la génération de celles-ci au niveau du réseau. L'assemblage de pages dynamiques suppose que la proportion du contenu cacheable soit importante, que les fragments cachés soient réutilisables et que ces derniers ne changent pas très souvent. Alors que la génération des pages au niveau du réseau suppose généralement que toute l'application est répliquable, ce qui n'est pas nécessairement vrai. Dans cette étude, nous proposons un CDN hybride basé sur la fragmentation, appelé FRACS, qui combine la fois l'assemblage de pages et la génération des fragments au niveau du réseau. FRACS répond aux besoins d'applications faiblement ou fortement dynamiques. FRACS identifie les fragments répliquables d'une application Web, ensuite il modifie le code de l'application en vue de prendre en compte les fragments définis et de permettre aux serveurs de générer les fragments séparément. De plus FRACS assure la consistance des données manipulées. Des expériences effectuées avec le benchmark TPC-W nous ont permis de réaliser jusqu'à 60% d'économies de bande passante et plus de 80% de réduction de temps de réponse.

## Mots Cles :

CDN, Applications Web, Contenu dynamique, cache et replication.

## I. INTRODUCTION

A Web application is an application which produces dynamic content through the use of server side processing. In this paper we are particularly interested in Web applications which are written in common scripting languages (PHP, JSP, ASP etc.), and which make use of a backend database. It goes without saying that the generation of dynamic content comes at a cost, a performance cost. In order to accelerate access to Web applications, content providers are increasingly turning to content distribution technologies, namely CDNs (Content Delivery Networks). A CDN is “a dedicated network of servers deployed throughout the Internet that Web publishers can use to distribute their content on a subscription basis” . The main benefit of a CDN is to enable rapid, reliable retrieval from any end-user location by replicating the content of customer Web sites and serving it to clients from nearby servers. Currently, CDNs serve the dynamic content in two major ways :

- Page Assembly : the page is constructed by aggregating its corresponding fragments on the edge servers. In this case, the edge servers do not generate the fragments, they simply cache those which have been generated by the origin server ;
- Edge Computing : the page is computed on the edge by replicating or migrating the application’s code (and possibly data) on the edge servers.

Both approaches have advantages and drawbacks and each one targets a certain class of applications. Reusing cached fragments of a page reduces its generation time as the overhead of generating the existing fragments is eliminated. But caching the fragments assumes that the proportion of cacheable content is high, that the cached fragments are reusable and that they do not change very often. For applications that do not meet these assumptions, replicating the application turns out to be a more effective alternative than caching. However, due to security and privacy concerns most web applications as well as their back-end databases (if any) cannot be cached entirely. Moreover, maintaining the consistency of all the application data is likely to be a significant problem.

Besides these limitations, current CDNs do not provide a means to scale the database component of a Web application. Hence, the CDN solution is not sufficient when the database system is the bottleneck [16].

To sum up, fragment caching is best suited to relatively static applications, while edge computing is more appropriate for highly dynamic applications. Nonetheless, it is seldom possible to replicate the whole application on the edge of the network, and to the best of our knowledge there is no tool which automatically determines the replicable code fragments of a Web application. Furthermore, an acceleration solution must provide a way to scale the database component.

In this paper, we propose a fragmentation-based CDN called FRACS, which combines both page assembly and edge computing in order to deliver the content efficiently to end users. For now, the fragmentation has only been considered from a cache oriented point of view. The defined fragments are usually HTML fragments which are meant to be stored on a cache server. We extend the notion of fragment so as to include two new types of fragments : application fragments and code fragments. We define the three categories of fragments as follows :

- ✓ *Code fragments* : a code fragment is a part of the application code.
- ✓ *HTML fragments* : an HTML fragment is the HTML code generated by a code fragment.
- ✓ *Application fragments* : an application fragment consists of a code fragment plus the data necessary to generate the HTML output.

The code and the application fragments allow the acceleration system to take advantage of application replication even when the application cannot be replicated entirely. FRACS transforms the standard monolithic Web applications into their corresponding fragmented version. It also provides the support to replicate the defined application fragments and to assemble the dynamic pages at the edge of the network. The replication only relies on the HTTP protocol. Most importantly, FRACS maintains the consistency of the manipulated fragments. FRACS only requires a small amount of human configuration. Actually, the administrators are asked to provide a list of sensitive data (expressed in terms of database tables) that they do not want to be replicated. Based on this information, our system fragments the application code and determines which fragments can be replicated along with the required database tables. In this study, we focus on E-commerce applications as they are data intensive and clearly illustrate the impossibility of replicating the whole application while making use of a significant piece of cacheable data.

FRACS consists in four logical modules : the adaptation module, the edge server module, the replication

manager module and the consistency manager module. The adaptation module is responsible for creating the fragmented version of the application and for creating the replication scripts. It is also responsible for extracting useful information which allows the system to handle the fragments correctly. The edge server module allows edge servers to replicate application fragments properly and serve them to end users. The replication manager module manages the whole replication process. It is responsible for replicating the application fragments as well as redirecting the clients to the appropriate replica. Finally, the consistency manager module provides the support necessary to maintain the consistency of the replicated tables and the stored HTML fragments.

The rest of the paper is organized as follows : Section II covers related work and highlights its advantages and drawbacks. Section III presents and discusses the proposed solution and the corresponding architecture. Section IV reports our experimental results. Finally, Section V summarizes our contribution and outlines the open issues.

## II. RELATED WORK

In order to address the shortcomings of total replication, it is necessary to be able to replicate the application partially. Hence, we first summarize the current approaches to fragmenting Web applications. Then we present the different systems which serve the content from the edge of the network. Finally, we describe the work related to scaling the database component of a Web application.

### A. Fragmentation of the application

As mentioned in the introduction, the existing fragmentation techniques are only meant to serve page assembly (and not edge computing) purposes. Assembling a page at the edge of the network implies that Web pages are broken down into fragments and that these fragments are separately cacheable. There is no standard method to fragment Web pages and generally if the fragmentation is not supported originally by the Web site, it is still often added manually. However, automatic fragmentation has been investigated. Related studies determine the fragments by periodically comparing different HTML output versions generated by the same URL ([15] and [12]). The main disadvantages of this approach are the overhead of fragmenting a page online and the redundant online processing.

Partial replication of the code needs other kinds of fragments. It requires splitting the code up into minimal replicable units which have complementary or common functions. Functionally speaking there are two ways to consider code partitioning : from a service oriented view point or from a logical view point. Each Web application consists of three logical layers and offers a certain number of services (see figure 1). A particular

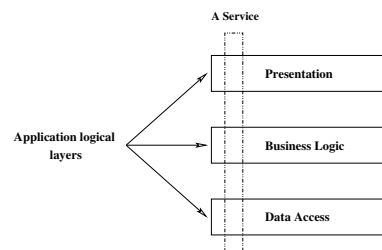


Fig. 1. Web application structure

service is composed of some functions belonging to the different logical layers. Service oriented partitioning consists in grouping together all the necessary code to execute a service, while logical partitioning consists in grouping all the code belonging to one specific layer. Thus one may replicate one or more services or one and more layers. To the best of our knowledge, there is no system which determines such fragments automatically.

### B. Serving the content from the network edge

1) *Page assembly*: The ability to assemble dynamic pages from individual HTML fragments means that only non-cacheable, non-cached or expired fragments need to be fetched from the origin Web server, thereby decreasing the load of the server and lowering the bandwidth consumption of the Web application.

The information on the fragments composing a page is usually piggybacked on the corresponding HTML code in a particular markup language. The markup language describing the fragments can be proprietary, however, a

standardization effort has been made to develop a uniform programming model to assemble dynamic pages. The resulting markup language called ESI (Edge Side Includes) allows developers to identify content fragments and specifies a content invalidation protocol for transparent content management across ESI-compliant solutions. A certain number of commercial CDNs have already implemented ESI language (Akamai [1]).

Akamai clients (content providers), use ESI to break the dynamic page into fragments with independent cacheability properties. These fragments are maintained as separate objects in the edge server's cache and are dynamically assembled into Web pages in response to user requests [7].

2) *Edge Computing*: When the dynamic content cannot be cached, it turns out to be more efficient to generate it on the edge of the network. This is commonly achieved by replicating the application on the edge servers. By generating the documents locally, the edge servers of a CDN reduce the amount of wide-area traffic, leading to a better client-perceived performance. Though edge computing does not impose a particular replication granule, most of the studies related to edge computing consider the whole application code as the replication granule. Moreover, existing studies generally focus on read-only applications.

An interesting study proposes an architecture, referred to as ACDN [18], which relies on the HTTP protocol and Web servers. Each ACDN server is a standard Web server that also contains a set of CGI scripts implementing ACDN-specific functionalities. Using the different scripts, a server monitors its own load and decides whether to send a replication request to a *Global Central Replicator* which keeps track of application replicas in the system. ACDN implementation is based on the concept of metafile. A metafile consists of two parts : the list of all the files comprising the application together with their last-modified dates ; and the initialization script that the recipient server must run before accepting any request. The main shortcoming of ACDN is that it solely considers read-only web applications. Moreover the metafiles describing the replicable parts of the application are produced manually.

### C. Scaling the database component of a Web application

There are two major ways to scale the back end database : either by replicating the database (totally or partially), or by caching the result of database queries and reusing them.

1) *Replication of the database*: Middleware systems like C-JDBC [3] and Ganymed [17] propose replicating the database in a cluster of servers. The focus of these studies is to improve the throughput of the underlying database within a cluster environment. Other studies ([21]), focus on improving the client perceived-performance and reducing wide-area update traffic. [21] proposes a system called GlobeDB for replicating read/write Web applications. GlobeDB automatically replicates a part of the application database and its relative access code. The underlying principle behind GlobeDB is to place each data unit only where it is accessed in order to facilitate consistency management. The replicated data units are clusters of database records calculated periodically based on the access statistics that are collected from the different edge servers. GlobeDB has two main disadvantages. On the one hand, it replicates all of the application's code at all replica servers. On the other hand, GlobeDB can only handle *simple queries*<sup>1</sup>, all complex queries (matching more than one row) are forwarded to the origin server which incurs a wide-area latency. GlobeDB provides *sequential consistency*. A system is defined to be sequentially consistent if the result of any execution of possibly concurrent operations is the same as if those operations were executed in some sequential order and the order in which they appear in the transaction is preserved [13]. In order to serialize the concurrent updates, each data cluster is assigned a master server which is the only server capable of updating the cluster. Any update to the cluster is immediately forwarded to the master of this cluster , which processes the stream of update requests and propagates the result to replicas. This reasoning only holds if the update operations are performed on only one data unit at a time.

2) *Caching the database query result*: Several commercial database caching systems (DBCACHE [2] and MTCACHE [14], oracle Oracle9i Database Cache [20], Tangosol [22]), propose caching the result of selected queries and keeping them consistent with the underlying database. The query result is typically stored on the origin server or on a cluster of servers.

Data-intensive Web applications can also use a third-party Database Scalability Service Provider (DSSP) which caches application queries results and answers queries on their behalf. The challenge of this solution is to ensure the consistency of the stored results when the database is updated. To address the problem, several

<sup>1</sup>a simple query is defined as a query based on primary keys of a table or that results in an exact match for only one record

systems ([8],[9],[10]) require the backend database server to track the contents of the proxy caches and forward relevant updates to each proxy server. Another system called S3 [11], proposes a distributed consistency management infrastructure that uses a publish/subscribe support to propagate update notifications. In a topic-based publish/subscribe system, a client receives all publications to each group to which it is subscribed. S3 users connect directly to proxy servers which store materialized views of query results instead of centralized home servers. When a dynamic Web application issues a database query, the proxy responds immediately using its database cache if possible. If the query result is not stored in the cache, the proxy forwards the request to the centralized database server and caches the reply. The proxy cache subscribes to some set of multicast groups when it caches a query, and broadcasts to some set of multicast groups for each update it issues. S3 proposes two strategies to associate database requests with topic-based multicast groups :

- associating multicast groups with the data on which queries depend,
- associating multicast groups with the data which updates affect.

The main disadvantage of S3 is that the current version of the system requires manual inspection the application's queries templates and definition of the multicast groups.

### III. FRACS SOLUTION

In order to improve the performance of Web applications, we propose a system called FRACS, which transforms a monolithic Web application into its corresponding fragmented version and which provides support for replicating the defined fragments on the network edge. Besides, FRACS provides the support for caching the generated HTML fragments whenever this is possible. This allows us to handle relatively static applications as well as more dynamic ones. Most importantly, FRACS maintains the consistency of the manipulated fragments.

Figure 2 depicts the global architecture of the system. FRACS consists in four logical modules : the adaptation

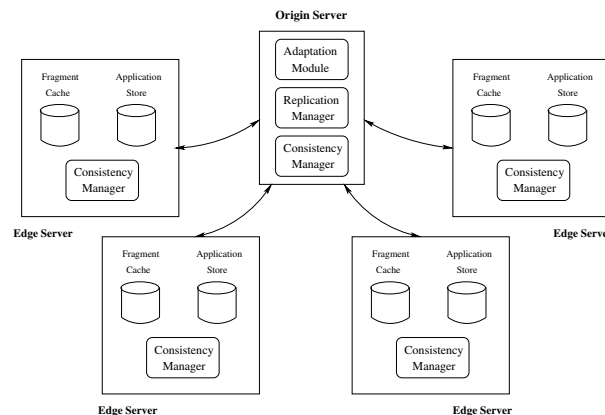


Fig. 2. System Architecture

module, the edge server module, the replication manager module and the consistency manager module. The adaptation module determines the fragments of a Web application that can be served from the edge, then it transforms the application's code so as to enable the server to generate fragmented pages and to support individual manipulation of fragments. It also creates replication scripts that automate the effective replication of application fragments.

The end users usually arrive on the origin server the first time they make a request. The replication manager module which is colocated with the origin server determines which edge server should process the request and redirects the users to this server. The replication manager is responsible for creating new replicas in the system. An edge server may host fragments of the application and have a local cache of fragments. In the current implementation, we chose to replicate all the replicable fragments of the application at once.

Finally, as its name suggests, the consistency manager manages the different aspects related to consistency, the latter uses a distributed approach to propagate the updates in the system.

#### A. Adaptation Module

The adaptation module is responsible for transforming the application in order to enable it to generate fragmented pages in ESI format and to support separate generation of fragments. This module performs three

main actions. First, it identifies the interesting code fragments which can be generated individually and collects metadata characterizing them. Second, based on some configuration data the adaptation module modifies the code in order to generate fragmented pages. Finally, this module identifies the replicable fragments and creates the scripts which will be used to replicate them. The adaptation module performs offline tasks so that the overhead of the adaptation does not impact the user's observed response time.

1) *Fragmentation and metadata collection*: The first challenge of the adaptation module is to fragment the Web pages efficiently and at a minimal cost. For Web applications that originally support the fragment handling, the problem does not arise. But when the sites are not originally fragmented some processing must be done in order to identify the fragments that will be cached and to enable the servers to generate them separately.

The adaptation module considers the independent scripts of a page as separate code fragments (which is equivalent to service oriented partitioning). This choice ensures that the different code fragments can be executed independently of one another with little modification to the cost. The HTML fragments produced by these code fragments represent interesting candidate fragments which are likely to be reused by the caches. The fragmentation system is a complex system (see [4] for details). The basic idea of the fragmentation system is to parse the application code statically rather than the HTML output. Static analysis avoids superfluous online processing and facilitates the identification of the fragments and their characteristics. The fragmentation system comprises a parser which extracts information from the code. It goes without saying that in order to handle a Web application, the fragmentation system must have the parser corresponding to the programming language of the Web application in question. The fragmentation system is designed to support the integration of new parsers, however the current implementation only comprises a PHP parser.

Firstly, the fragmentation system collects information from the code. Static analysis of the code facilitates the metadata extraction as the code draws a clear separation between the static and the dynamic content and contains the exact set of variables that are actually used and the exact set of database accesses (unlike the output pages where the code has already been executed and the HTML generated). We mainly gather the following information from the code :

- the variables and the functions used by the scripts,
- the beginning and the end offsets of the scripts,
- the database queries made by the scripts.

The variables used by a script consist in HTTP and CGI elements in general (commonly the GET method arguments and the cookies) and the programmer's defined variables. On the one hand, the variables and functions used by the scripts allow us to find the independent scripts (code fragments) of a particular page. On the other hand, these variables help us to characterize the code fragments efficiently so as to increase the cached fragments reuse rate (see section III-B).

The beginning and the end offsets are used to locate the fragments position inside a page.

Finally, the database queries are useful for the consistency management.

2) *Transformation of the code*: In order to generate fragmented Web pages, particular tagging instructions are inserted at the beginning and at the end of the identified code fragments. When the code of the dynamic page is executed, the tagging instructions result in surrounding the HTML fragments by ESI tags. The ESI tags are not only used to demarcate the fragments but also to describe their attributes. The attributes mainly consist in the name of the fragment, the variables that it depends on and some data characterizing the database dependencies.

The adaptation module also inserts into the code particular functions which capture the queries made by the fragment, and changes the hypertext links (to the origin server) accordingly so as to point to the local host. This is done via a variable which is replaced by the appropriate value at the runtime.

Finally, in order to make a code fragment separately fetchable the adaptation module reifies it by stripping its body from the page and storing it as a separate program, then it replaces its occurrence with the appropriate inclusion instruction. Henceforth, a dynamic page will be assimilated to a template containing references to underlying fragments.

3) *Replication scripts creation*: FRACS replicates application fragments (i.e. code fragments plus the accessed tables). The code fragments could have been replicated without the corresponding backend database (see figure 3), the edge servers would be able to generate the pages near the clients, but the database requests would still have to be executed on the origin server database.

A study [23] has demonstrated that this configuration is worse than the traditional centralized architecture

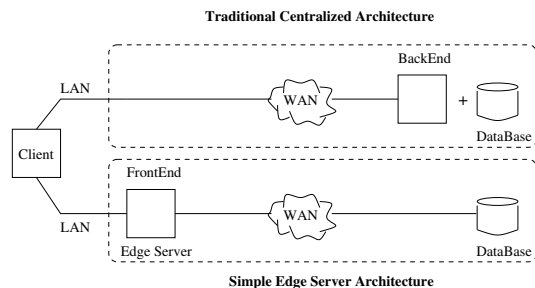


Fig. 3. Configuration

because client requests to the edge servers often trigger multiple requests from the edge server to the origin server across the WAN. Therefore, it is more interesting to replicate the code and the database together as long as consistency is maintained. Additionally, replicating the database tables makes it possible to scale the database component of a data-intensive Web application. The main problem is then to determine the fragments to replicate. The code fragments identified in the previous step are good candidates for the replication. In order to decide whether a code fragment is replicable, the adaptation module identifies the tables accessed by the code fragment then it compares them against the unreplicable tables specified by the administrator. If all the accessed tables are replicable the code fragment is marked replicable, otherwise it is marked unreplicable. In order to replicate an application fragment, an edge server needs to copy the fragment code, to create the required database tables and to fill them with the appropriate data. As we will see in section III-D, the replication decision is made by the replication manager. The replication manager has to send all the data necessary to replicate the application fragment as well as the replication script to the edge server. The replication manager itself uses another replication script to perform the actions above. The adaptation module automatically creates the different replication scripts which will be used by the replication manager and the edge servers to create and delete the application fragments. For this purpose, the adaptation module reuses some of the data collected during the static analysis, namely : the path of a code fragment and the database tables used by the latter. It should be noticed that the adaptation module has the creation scripts (SQL instructions) of all the replicable tables. For example, figure 4, represents the replication manager's script used to collect the data necessary to

```
#!/bin/bash
(1) mysql -u ikram --password=ipass << EOF
    use tpcw;
    SELECT * FROM item INTO OUTFILE 'item.dat'
    EOF
(2) mv item.dat DATA
(3) cp ~/www/TPCW/schemas/tpcw_item_schema.sql DATA
(4) cp ~/www/TPCW/search.php DATA
(5) cp ~/www/TPCW/repscripts/createsearch.sh DATA/
    cp ~/www/TPCW/repscripts/deletesearch.sh DATA/
(6) cp ~/www/TPCW/conf/search.cnf DATA/
(7) tar cvf DATA.tar DATA
```

Fig. 4. Replication Manager Shell Script

replicate the the code fragment “search.php” which uses the table “item”. As we can see in the figure, the first set of instructions (1) copies the content of the table “item” into a file named “item.dat”. Then (2) moves this file into a directory which is to be compressed and sent to the client. (3), (4) and (5) respectively copy the script which contains the sql code to create the table “item”, the code of the fragment to replicate and the replication scripts which will be used by the edge server in order to create and delete the application fragment. Instruction (6) copies the file containing the configuration data of the fragment (search.cnf). This file mainly contains the address of the replication manager and the pertinence criteria of a replica. Finally, instruction (7) compresses the directory and creates the tarball which will be sent to the edge server. Figure 5, details the

script used by the edge server so as to replicate code fragment search.php (i.e. createsearch.sh). The first set of

```
#!/bin/bash
mysql -u edgename --password=edgepass << EOF
create database tpcw;
use tpcw;
(1) source tpcw_item_schema.sql
LOAD DATA INFILE 'item.dat' INTO TABLE item;
EOF
(2) mkdir -p ~/www/TPCW/conf
(3) cp search.php ~/www/TPCW
(4) cp search.cnf ~/www/TPCW/conf
```

Fig. 5. Edge Server Shell Script : createsearch.sh

instructions (1), creates the database tpcw, then uses the script tpcw\_item\_schema to create the item table, and finally loads the data contained in the file “item.dat” into the item table. (2) creates the application directories. (3) and (4) copy the code fragment and the configuration file into the appropriate directories. Figure 6, depicts the script used to delete the application fragment. The script simply drops the created tables and removes the code fragment.

```
#!/bin/bash
mysql -u edgename --password=edgepass << EOF
(1) use tpcw;
DROP TABLE IF EXISTS 'item';
EOF
(2) rm -f ~/www/TPCW/search.php
```

Fig. 6. Edge Server Shell Script : deletesearch.sh

## B. The edge server module

Edge servers are simply servers of the CDN which are located on the edge of the network. Each edge server has a local store of applications, a cache of fragments and a consistency manager. In FRACS, an edge server may host a replica of the application either temporarily or permanently, and has a local cache of HTML fragments.

FRACS needs to collect some measurements on the edge servers, for efficiency and scalability concerns, edge servers monitor their own state.

1) *Replication of the application fragments:* When the replication manager decides to replicate fragments of the application on a target edge server, it sends a POST query to a particular program on the target edge server. This program is responsible for uncompressing the request body and running the embedded replication script. As mentioned previously, in the current implementation we chose to replicate all the replicable fragments of the application at once, rather than separately. Once the replica has been installed, the edge server starts receiving user requests. Before servicing a user, an edge server has to fetch the session objects related to this user (see section III-D for details). When the content requested by the end users is replicable, the edge server generates it locally. The edge servers can make both *selection* queries (SELECT) and *modification* queries (UPDATE, DELETE, INSERT INTO) to the local database. When a locally generated fragment makes a request to the database, this request is automatically detected by the instructions inserted into the code by the adaptation module, and sent to the consistency manager. The consistency manager is responsible for propagating the updates so as to maintain the consistency of the replicated tables.

2) *Cache of HTML fragments:* The edge server has a cache of fragments which caches the locally generated HTML fragments. It is important to note that the HTML fragments can be cached by any ESI compliant server, nonetheless, in this paper we only focus on the caching of locally generated HTML fragments.

The cache uses the templates of the dynamic pages in order to cache and reuse the fragments. The template of a page describes the different fragments belonging to the page as well as their attributes. The template is

described by the ESI tags inserted into the code by the adaptation module. When an edge server generates a dynamic page for the first time, it analyzes the HTML generated so as to find metadata of the page, then caches the different fragments. Upon receiving a second request for the page, the edge server checks the corresponding template, fetches or generates the missing fragments and reconstructs the page. In order to take full advantage of the cached content, an edge server should only regenerate the missing and stale fragments for every new request. This implies that the edge server knows in advance about the fragments it is supposed to return when it receives a request. This is not easy to achieve unless the exact URL has already been requested. We address this limitation by introducing what we call *filters*. A filter is a piece of information associated with a script which is used to produce a unique identifier per HTML fragment produced by the script. Generally, the attribution of an identifier to an HTML fragment is done after the generation of the page containing the HTML fragment in question. The filters make it possible for an edge server to calculate in advance the identifiers of fragments that should be present in a page provided that the template of the page has already been accessed. Actually, the output of a code fragment depends on a set of variables defined in the environment in which the corresponding script is running. Naturally, it also depends on the content of the database (if the displayed information is retrieved from a database), but updating the database does not affect the key of fragment, it can at most render the fragment stale. Let us take a simple example to illustrate the principle of filters. Figure 7 represents a page containing two fragments whose output respectively depend on param1 and param2, passed to the program via the query string. The filter associated with a code fragment contains the identifier of the code fragment and the

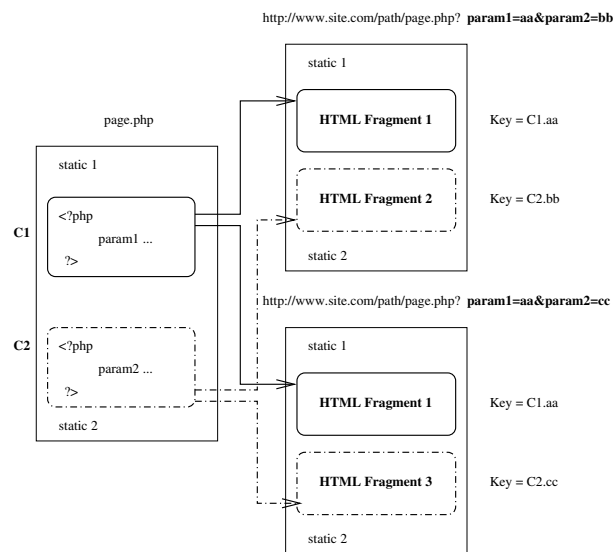


Fig. 7. Filters

variables that the HTML fragment depends on. In this example, say that “C1” is the identifier of the first code fragment and “C2” is the identifier of the second fragment. Then, the identifier of the second fragment contains “C2” and “param2”. If we assume for the sake of simplicity that the HTML fragment key is the concatenation of the filter’s fields, then the HTML fragments produced by the respective requests to the URLs :

`http://server/page.php?param1=aa&param2=bb`

and

`http://server/page.php?param1=aa&param2=cc`

will be identified by the respective keys : “C2.bb” and “C2.cc”. These keys can be calculated provided that the template is already stored. Therefore, the HTML fragment produced by a code fragment can be reused to assemble any other page containing the same fragment (example the fragment identified by “C1.aa”).

The cache of HTML fragments introduces the constraint of maintaining the consistency of the stored content. This issue is addressed by the consistency manager.

3) *Monitoring*: For scalability concerns, each edge server monitors its own parameters and sends periodical reports to a central monitor in the CDN. All the edge servers measure their load and send it to the central monitor. The load of a server is estimated by the number of active connections. Moreover, each edge server which replicates the application compares the current load to a threshold contained in the configuration file

sent with the replication script. If the load is superior to the threshold, the edge server sends an alert to the monitor, which in turn sends it to the concerned replication manager. Each edge server which has a replica of the application also measures the amount of data served and the amount of update data received during the last period. These measurements are used locally to check the relevance of the replica. In our study, the relevance of a replica is estimated by the bandwidth saving it achieves. The edge server compares the volume of updates received against the volume of data served. If the volume of updates is greater than the volume of data served, it means that the cost of maintaining a replica up to date outweighs its benefits. Nonetheless, the edge server cannot delete the replica immediately, it has to send a deletion query to the replication manager which decides ultimately whether or not to delete the replica.

### C. Consistency Manager Module

The consistency manager is a complex system, its internal logic is beyond the scope of this paper, however we will give a brief overview of its functioning and complexity. The consistency manager has two main functions : to maintain the consistency of the replicated databases and to ensure the freshness of the stored HTML fragments.

1) *Consistency of the replicated database tables*: Based on how strictly the databases are kept consistent, algorithms can be classified into two categories : *strong database consistency* and *weak database consistency*. Weak consistency is the model in which the modifications are not propagated instantly, and hence, different edge servers may work on different versions of the same database. On the other hand, in a strong consistency model every modification query must be acknowledged. A server will not update an object until it receives all the acknowledgement messages from the edge servers which are supposed to apply the modifications. Instead of sending the modifications instantly to the edge servers, we propose logging these requests and sending them periodically. A priori, sending modification queries in batch ensures weak consistency, however we choose a small period so that the data is not older than a certain threshold. The threshold used in the experiments with TPC-W is one minute. The consistency manager logs the modifications made locally during the specified period, and sends them to the replication manager which multicasts them to the edge servers which replicate the tables involved. Upon receiving the set of updates, the edge servers simply apply them to the local database. For now, we do not check whether the updates are conflictual.

2) *Consistency of the cached HTML fragments*: The HTML fragments may become stale for two reasons : either the code fragment which generated them has changed, or the database tables accessed by the corresponding code fragments have been modified. In this study we assume that the code of the application is not modified (though as we will explain in section V, modifying the application code is not a major hurdle). In order to invalidate the cached HTML fragments we associate to a fragment its database dependencies when the fragment makes a selection query to the database, then we check whether a modification query invalidates the locally stored HTML fragments. As explained previously, the local selection and the modification queries are captured by functions inserted into the code by the adaptation module and sent to the local replication manager along with the identifier of the fragment which issued them. When the replication manager receives a selection query it translates the latter into its relative Disjunctive Normal Form (DNF). After that, it calculates the MD5s of the resulting clauses and stores the mapping between the fragments identifiers and the MD5s of the clauses that they depend on. Then, the different clauses of the DNF are inserted into a *predicates tree* whose leaves are the previously calculated MD5s. The predicates tree is designed to facilitate the comparison of the stored clauses against a new literal. Finally, it subscribes to the notification server for these MD5s. When a modification query is sent to the consistency manager, the latter translates it as well into its corresponding DNF, then it traverses the tree following the matching branches so as to find the MD5s of invalidated clauses. An update notification is sent to the notification server which is responsible for the MD5s. It must be noted that it is not usually possible to decide whether a modification invalidates a fragment or not (unless additional requests are made to the database), in such cases we adopt the pessimistic approach by invalidating all the suspect fragments. The algorithm which determines the fragments to invalidate has a sublinear complexity :  $c(n) = O(n^p)$ , where  $p$  is comprised between 0 and 1.

### D. The replication manager module

The replication manager has to redirect the clients, administer the replication process and keep track of the application replicas in the system. There is one replication manager per application, it is colocated with the

origin server.

1) *Redirection of the clients*: The replication manager receives requests from both end users and edge servers. The clients usually arrive on the origin server at first independently from the requested content. As concerns the edge servers, they only request a fragment from the replication manager when the fragment is not replicable because they replicate all the replicable fragments of the application at once.

When the requested content is not replicable, the request is directly transmitted to the origin server. Otherwise, the replication manager redirects the client to any nearby edge server which has a replica of the application and which is not overloaded. The redirection is made via the 307 HTTP status code (temporary redirect), so that the client is compelled to turn back to the replication manager for the next session. If all the edge servers near the client are overloaded the replication manager may create a new replica on the least loaded edge server in the neighborhood.

2) *Replication of the application*: When all the replica servers in the neighborhood of a client are loaded, the replication manager checks whether it is possible and relevant to create a new replica of the application, otherwise it simply distributes the requests evenly among the loaded replica servers. Each new replica spawns new traffic in order to maintain the consistency of all the replicas in the system. Therefore, the system should offer a reasonable tradeoff between performance improvement and consistency management. We introduce a maximum number of replicas in order to make sure that the bandwidth consumed by the creation of a replica and the consistency messages between the existing replicas is never larger than a certain proportion of the overall traffic. This maximum number of replicas is calculated based on the average proportion of the traffic we are willing to dedicate to the creation and the maintenance of the replicas.

When the replication manager decides to replicate the application, it queries the central monitor of the CDN in order to locate the candidate target edge servers in a particular region. Then, the replication manager runs the appropriate replication script which produces a compressed file comprising all the necessary data for the replication. This file is sent in the body of a POST request to a program residing in the target edge server. Thence, the replication only relies on the HTTP protocol.

The replication manager receives deletion queries from the edge servers in order for them to have permission to delete their replicas. This helps to limit the number of replicas in the system. If the number of existing replicas is near the maximum number of replicas, the replication manager agrees to delete the replica.

3) *Session management*: End-user session state is a common aspect of E-commerce applications. As pointed out by a recent study [6], the dynamic mapping of users to the appropriate application instance across the network introduces the problem of “session affinity”. Having one centralized database to provide persistent session storage would add unnecessary latency in servicing the request and might even cancel out the benefits of moving an application to the edge. In FRACS, the replication manager allows client session objects to be replicated across edge servers in the network. The first time a user is mapped to an edge server, the server asks the replication manager whether there is a session object for this user. If the session object exists, the replication manager returns the identifier of the last server which has requested this object (and presumably modified it) and thus which has the latest version of it. Otherwise a new session object is created for the user on the edge server and is cached locally. The replication manager records the identifier of this server as the latest server to have requested that particular session object. The session objects are set to expire after a specified TTL.

#### IV. EXPERIMENTS

We chose the TPC-W benchmark in order to evaluate the benefits of FRACS. TPC-W specifies an E-commerce workload that simulates the activities of a retail store web site. Users are emulated via several Remote Browsers Emulators (RBEs). Emulated users can browse and order products from the site. The benchmark defines two categories of Web interactions : browsing and ordering. Browsing interactions include displaying the home page, searching for an item, viewing product details, etc., while ordering interactions include the more resource intensive ones such as buying an item, updating items in the shopping cart, displaying an order, etc. The RBEs can be configured to generate different interaction mixes :

- Browsing mix : 95% browsing and 5% ordering,
- Shopping mix : 80% browsing and 20% ordering,
- Ordering mix : 50% browsing and 50% ordering.

The schema defined in the specification of TPC-W consists of eight tables : customer, address, order, order line, credit card transaction, item, author and country. We only consider three tables replicable : item, author

and country.

In our experiments we mainly study the response time because it is the parameter which is usually, implicitly, used by clients to evaluate the performance of an E-commerce application.

The four modules of FRACS are all implemented but have different levels of maturity. The adaptation module, is fully implemented and tested with other PHP benchmarks, while the other modules have only been tested with TPC-W. In addition, the current version of the system uses a simulation of a central monitor which relies on a static distribution of the clients and the edge servers.

Figure 8 depicts the test bed. We deployed our prototype across eight similar edge servers. The servers run

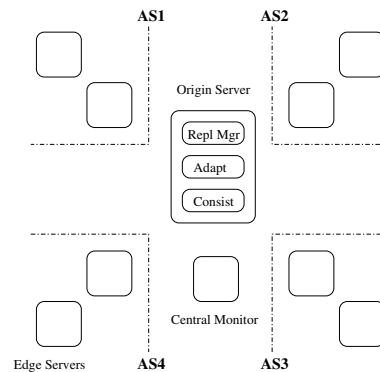


Fig. 8. Test Bed

on a Pentium 4, 3.6GHz with 1GB of memory. Each edge server has a local cache of fragments and uses Apache2-2.0.48 Web server with PHP 4.3.4. It also uses SQL 4.0.18 as a database server. The different servers are connected to each other through a 100Mbps Ethernet switch. We virtually partitioned the edge servers into four Autonomous Systems. The propagation delays between the autonomous systems, the origin server and the replication manager are tunable but are of course fixed at the beginning of any experimentation. In particular, we keep the propagation delays unchanged if we have to compare the results of several experiments.

First of all, in order to have an idea of the effectiveness of the acceleration we compared the performances of our solution, on the one hand, with the basic configuration (i.e. with no acceleration tool), and on the other hand with the configuration where a simple cache of HTML fragments is deployed. In the first configuration, the edge server forwards all the requests to the origin server. In the second configuration, the edge server assembles the dynamic page by caching and reusing the cacheable fragments, and when the fragments are not stored or cannot be cached the requests are forwarded to the origin server. The third configuration corresponds to FRACS.

In this experiment, the measurements were made on the edge servers. The response time is calculated as the difference between the moment a request is made to the edge server and the moment when the edge server sends back the response to the client. This response time approximates the response time observed by the end users as they are served by nearby edge servers.

Figure 9 presents the average response time for the shopping mix workload with the different configurations.

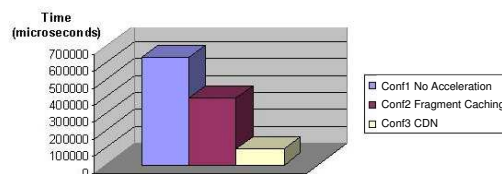


Fig. 9. Average response time

As we can see in this figure, our solution greatly improves the response time even if compared with caching. The average response time proves the efficiency of the solution, however the instantaneous response time illustrates more clearly the origin of the acceleration.

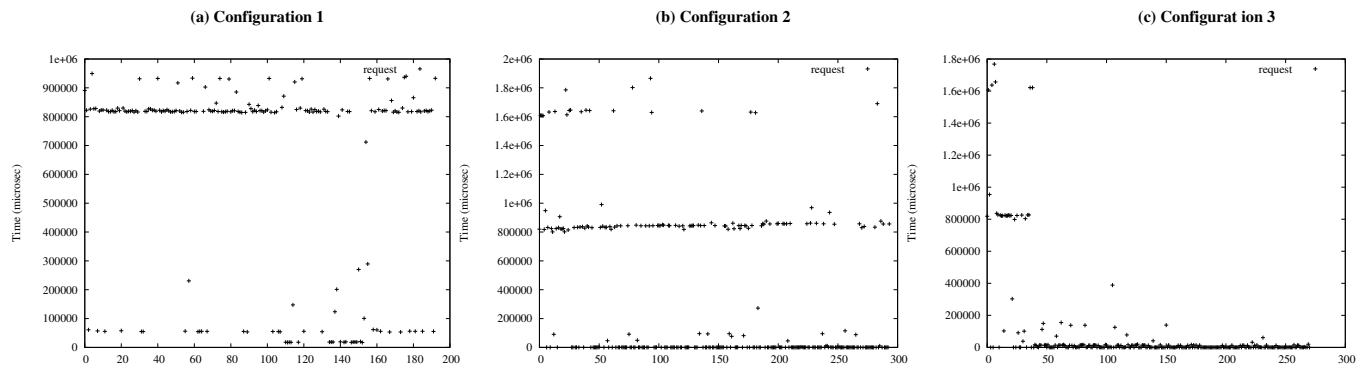


Fig. 10. Instantaneous response time

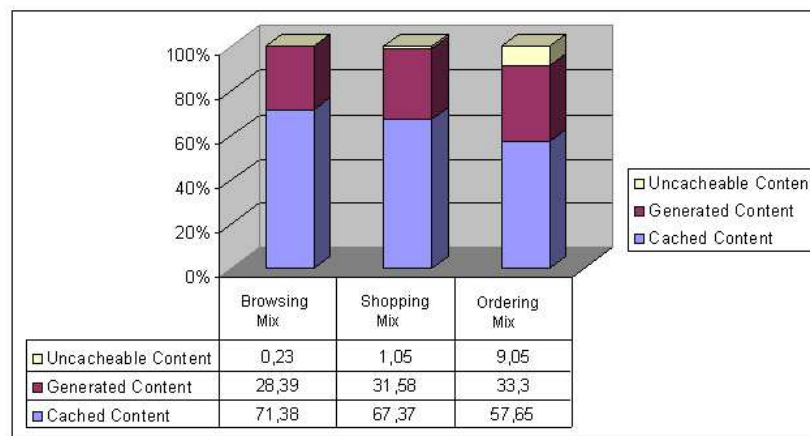


Fig. 11. Content classification

Figure 10 draws a parallel between the instantaneous response time of the three configurations. The reader's attention should be drawn to some details which may be misleading at first glance. For instance, one would expect the origin server's response time to be constant throughout time in configuration 1 (as no acceleration mechanism is deployed), however we observe that the response time sometimes decreases tremendously. This is due to some caching mechanisms used by both SQL and Apache servers. As concerns the second and the third configurations, we notice, in contrast, that the response is sometimes very high. This is due to a shortcoming of our current implementation. In fact, we only developed a basic proxy for our experiments which closes the connections after every request. This is particularly penalizing when a large number of missing fragments have to be fetched from a remote location.

Finally, we can see that the response time rapidly decreases in configuration 2 for a certain number of requests thanks to the stored fragments. The same is true for configuration 3 and we can clearly observe in figure 10 (c) the moment when the edge server has obtained a fragment of the application. Our system also diminishes the bandwidth consumption of the Web application insofar as it reuses the cached HTML fragments, whenever this is possible, and it generates the fragments locally or on nearby servers otherwise. Figure 11 and figure 12 respectively depict the average percentage of the reused content for the three different workload mixes and the corresponding reduction in response time. We can clearly observe that the percentage of the reused content is very high for all the workload mixes. This stems from the fragmentation method we are using (a thorough description of the fragmentation module can be found in [5]).

However, our system generates some traffic in order to maintain the consistency of the stored content. In our experiments with TPC-W we fixed the period to propagate the updates to 5 minutes. But even for the ordering mix workload, the bandwidth consumed by the update messages was barely 3% of the bandwidth consumed

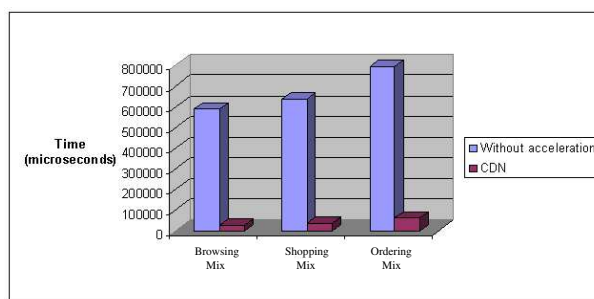


Fig. 12. Response time savings

by the application.

## V. CONCLUSIONS AND FUTURE WORK

In this paper we propose a hybrid CDN called FRACS which transforms a standard Web application into its corresponding fragmented version. FRACS determines and deploys the fragments of an application that can be served from the edge. The system we propose combines both page assembly and edge computing in order to deliver the dynamic content efficiently from the edge of the network. FRACS reduces the E-commerce applications response time as well as bandwidth consumption by replicating the application fragments at the edge of the network and caching them whenever this is possible. Moreover FRACS maintains the consistency of all the manipulated data. We have evaluated the performances of the system using the TPC-W benchmark and the results we obtained were very satisfactory.

In order to validate our approach on different types of applications, we are currently investigating the RUBiS benchmark [19] which is an auction site prototype modeled on eBay.

There are also some issues which have not been addressed in this paper and which we want to investigate as future work. For instance, as we said in section III, we assume that the code fragments are not modified. The modification of the application programs is likely to be done by a human administrator on the origin server. Thus we can simply introduce a new module through which the administrators can execute their modifications. This module would be a sort of wrapper which applies the modifications and which triggers invalidation messages for all the code and HTML fragments concerned by the modifications.

For now, we replicate all the replicable fragments of the application at once, it should be interesting to try to define clusters of application fragments which for instance minimize the number of exchanged update messages between the different replicas. We also implicitly assume that the simultaneous database updates on the different edge servers are not conflictual ; it would be more realistic to consider some mechanisms to handle this potential problem. Finally, in order to evaluate the real benefits of the approach, it would be pertinent to compare our results with some existing systems.

## REFERENCES

- [1] <http://www.akamai.com>.
- [2] C. Bornhvd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptative database caching with dbcach. *Data Engineering*, June 2004.
- [3] E. Cecchet. C-jdbc : A middleware framework for database clustering. *Data Engineering*, June 2004.
- [4] Ikram Chabbouh and Mesaac Makpangou. A configuration tool for caching dynamic pages. In *In proceedings of the 9th Int Workshop on Web Content Caching and Distribution*. WCW4, October 2004.
- [5] Ikram Chabbouh and Mesaac Makpangou. Caching dynamic content with automatic fragmentation. In *In proceedings of the 7th Int Conference on Information Integration and Web-Based Applications and services, Kuala Lumpur, Malaysia*. IIWAS2005, September 2005.
- [6] Andy Davis, Jay Parikh, and William E. Weihl. Edgecomputing : Extending enterprise applications to the edge of the internet. In *In proceedings of the WWW 2004*. ACM, May 2004.
- [7] John Dille, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, October 2002.
- [8] K. Amiri et al. Dbproxy : A dynamic data cache for web applications. In *ICDE*, 2003.
- [9] Q. Luo et al. Middle-tier database caching for e-business. In *SIGMOD*, 2002.
- [10] W. Li et al. Cache portal ii : Acceleration of very large data center-hosted database-driven web applications. In *VLDB*, 2003.

- [11] Charles Garrod, Amit Manjhi, Anastassia Ailamaki, Phil Gibbons, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. Scalable consistency management for web database caches. Technical report, Carnegie Mellon University, Intel Research Pittsburg, Akamai Technologies and Yahoo Research, 2006.
- [12] Vaggelis Kapoulas Ioannis Misedakis and Christos Bouras. Web fragmentation and content manipulation for constructing personalized portals. *APWeb 2004, LNCS 3007*, 2004.
- [13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, September 1979.
- [14] P. Larson, J. Goldstein, and J. Zhou. Mtcache : Mid-tier database caching for sql server. *Data Engineering*, June 2004.
- [15] Lakshmir Ramaswamy Arun Iyengar Ling Liu and Fred Dougliis. Techniques for efficient fragment detection in web pages. In *Proceedings of the 12th International Conference on Information and Knowledge Management, CIKM 2003*, November 2003.
- [16] Amit Manjhi, Anastassia Ailamaki, Bruce M. Maggs, Todd C. Mowry, Christopher Olson, and Anthony Tomasic. Simultaneous scalability and security for data-intensive web applications. In *SIGMOD 2006, Chicago, Illinois, USA*, June 2006.
- [17] C. Plattner and G. Alonso. Ganymed : Scalable replication for transactional web applications. In *International Middleware Conference*, October 2004.
- [18] Michael Rabinovich, Zhen Xiao, and Amit Aggarwal. Computing on the edge : A platform for replicating internet applications. In *In proceedings of the Workshop Web Caching and Distribution*, september 2003.
- [19] RUBiS. <http://rubis.objectweb.org/>.
- [20] Oracle9i Application Server. <http://www.oracle.com/technology/documentation/ias.html>.
- [21] Swaminathan Sivasubramanian, Gustavo Alonso, Guillaume Pierre, and Maarten van Steen. Globedb : Autonomic data replication for web applications, 2004.
- [22] Tangosol. <http://www.tangosol.com>.
- [23] Chun Yuan, Yu Chen, and Zheng Zhang. Evaluation of edge caching/offloading for dynamic content delivery. In *ACM Proceedings of the WWW2003, Budapest, Hungary*, May 2003.