Updating Web Views Distributed over Wide Area Networks

Antonis Sidiropoulos and Dimitrios Katsaros

Department of Informatics, Aristotle University
54124 Thessaloniki, Greece
Email: {asidirop, dkatsaro}@csd.auth.gr

Abstract. The deployment of Content Distribution Networks (CDNs) to effectively disseminate frequently changing Web content — Web views — to large client populations introduces several interesting problems. One of the dominant issues in this setting is the schedule according to which the disseminating server will send the updated Web pages to the CDN's proxy caches. Thus, determining a server push schedule in order to minimize page staleness and server load is a vital issue in maximizing the scalability and usefulness of the CDN. In this paper, we study the problem of scheduling Web view updates in Content Distribution Networks. We devise an algorithm to constantly refresh the Web views, which accounts for the update rate, the popularity and the position of each Web view in the caches of the CDN. We present experimental evidence using synthetic data, which show that our algorithm consistently and significantly outperforms FIFO scheduling.

1 Introduction

Content Delivery (or Distribution) Networks (CDN) provide users with fast and Reliable delivery of Web content, streaming media, and transaction processing across the Internet. Several companies e.g., Akamai, specialize in providing secure, outsourced services and software. Content Delivery Networks provide solutions that optimize Web site performance, deliver broadcast-caliber streaming media, and provide interactive application services. For instance, Akamai Technologies maintains the largest CDN with more than 12,000 edge servers in more than 1,000 networks worldwide [8].

As content delivery pathways between the server and the user continue to become More congested, problems such as sites that load slowly, or crash during delivery are increasing. A content delivery network solves these problems while reducing infrastructure costs. Some recent research done by Akamai, has shown that Web sites using a CDN can increase click-throughs by 20%, reduce abandonment rates by 10-15%, and increase order completion by 15%. CDNs resolve performance problems related to Web server processing delays and Internet delays. Users requesting popular Web content may well have those requests served from a location much closer to them (a local Network Provider's data center), rather than from much farther away at the original Web server. By serving content requests from a server much closer to the user, a quality CDN can reduce the likelihood of overloaded Web servers and Internet delays. A CDN can deliver rich, compelling content while increasing customer loyalty and strengthening a company's profile.

Until recently, content delivery networks were typically focused on delivering Static content, but recently they can also accelerate dynamic and personalized content. For instance, Akamai's EdgeSuite is a next-generation content delivery service, leveraging its core technology and industry-leading content delivery network for the distribution of an entire Web site including static, dynamic, embedded objects, and HTML.

A key parameter to these distributed technologies is the timely dissemination of the updated material to the edge servers. The Web pages that change frequently in the origin Web server and are outsourced in the edge servers — referred to as *Web views* in [13] — must be maintained up-to-date in order to increase the user satisfaction by reducing the content staleness. Thus, the server must deploy a schedule to push the updates to the edge servers. This schedule should cope with view popularities and view generation costs.

1.1 Overview of our Approach

In this paper, we study the problem of best-effort cache coherence maintenance. We focus on stale caching environments with a large number of caches — refered to as edge servers in the CDN literature — whose content must be synchronized with the data residing in an origin Web server. We do not care to keep the cached objects *transactionally consistent* with the origin data (although this may happen), because of the complexity and the cost of the required protocols [20]. Furthermore, even propagating all updates in a nontransactional fashion may be infeasible, due to the huge data collections that are updated and the network or computational resources required. Thus, we focus on a *best-effort* coherence maintenance scheme. This scenario in depicted in Figure 1.

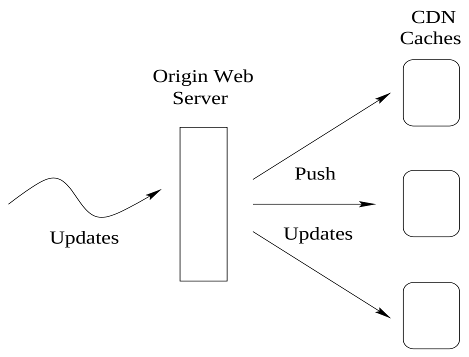


Fig. 1. Conceptual diagram of the problem.

We assume that each cache contains replicas of all the data objects of interest. (Thus, we do not consider Web cache replacement policies [12].) In our setting, the origin server is responsible for pushing each updated object to its caches and we do not consider cache-initiated prefetching [15].

In stale caching, the value of an object at the cache and origin may differ. This difference is called *divergence* and it can be measured using a number of possible metrics including Boolean freshness (up-to-date or not), number of changes since refresh or value deviation. The appropriate metric to use depends on the data and the application objectives. We define our metric in section 3. In spite of the metric used, the objective of the best-effort cache coherence maintenance is to minimize the (weighted) sum of the divergence values for each outsourced object.

If enough resources are available, i.e., high network bandwidth, low update rates, then it is possible to keep the objects transactionally consistent. Otherwise, we must prioritize some refreshes based either on the divergence value and/or a weighting scheme that reflects the relative importance of the updated objects. We describe our prioritization/scheduling scheme in section 4. In section 5, we show experimental evidence that our proposed scheme achieves superior performance in terms of minimizing staleness over a baseline scheme that implements a FIFO scheduling.

2 Related Work

A large body of research is related to the problem of maintaining the coherence of outsourced Web objects or as it is frequently called the *cache synchronization problem.* We outline the most important and most relevant work here.

Many stale caching and replication approaches have been proposed [20, 9]. However, all these approaches do not consider environments in which there is not enough bandwidth to propagate all updates.

In the approaches presented in [5] and [10] the cache plays the central role. It tries to predict which objects have changed and by how much. If the objects do not change in regular predictable intervals then the refresh schedule delivers poor quality to the users. Contrary to these approaches, we do no try to make predictions about object update rates, but we are aware of the exact time the object updates.

A lot of work considered the problem of minimizing the bandwidth consumption and query latency in the presence of constraints on the age or accuracy of cached objects, see for instance [17, 18, 7, 6, 1]. Our work differs from theirs mainly in that we do not consider any constraints upon the cached data.

Some other work [3, 2, 16] considered the issue of sending out invalidation messages to caches but did not investigate refreshing the updated objects in the background as we do.

Our work bears the largest similarity with the work reported in [4, 14, 19, 22]. Challenger et al., in [4] deal with maintaining strong cache coherence, whereas we focus on weak consistency. Labrinidis and Roussopoulos in [14] focus on a system composed of a content cache and several sources (base relations). It is assumed that the sources and the cache are connected over a local area network and thus they do not consider the unpredictable nature of the Internet. Olston and Widom in [19] present an approach where the cache cooperates with the sources in the formation of the schedule and thus it is different from our approach which is based solely on a push-based fashion. The model employed in [22] is alike ours, but the focus of their work is to determine the optimal transmission of object updates under some fixed assumptions about object update rates and transmission capabilities of the server.

Finally, our work is loosely related to the work done on scheduling events in real-time systems [21]. Though, these approaches assume strict completion deadlines and their objective is to minimize the percentage of events that miss their deadline.

3 Object Staleness

Let us consider an object O residing in the origin site, which undergoes updates over time. Let $C(O)$ represent a cached copy of O. Let also $V(O, t_1)$ be the value of O at time t_1 and $V(C(O), t_1)$ be the value of $C(O)$ at time t_1. In general, the *divergence* between O and $C(O)$ at time t_1 is given by a numerical function $D(O, t_1)$. Immediately after a refresh, the value of this function becomes zero (assuming zero propagation time), but its value may become greater than zero between refreshes. Depending on the application, we can use different divergence functions. In this work, following the approaches in [5, 14], we adopt a Boolean model for the divergence function. That is, $D(O, t_1) = 0$ iff $V(O, t_1) = V(C(O), t_1)$ and $D(O, t_1) = 1$ iff $V(O, t_1) \neq V(C(O), t_1)$ at time t_1. We call *freshness* the reverse of the divergence, that is, *freshness* $= 1 -$ *divergence* or $F(O, t_1) = 1 - D(O, t_1)$. Figure 2 illustrates the evolution of the freshness of an object.

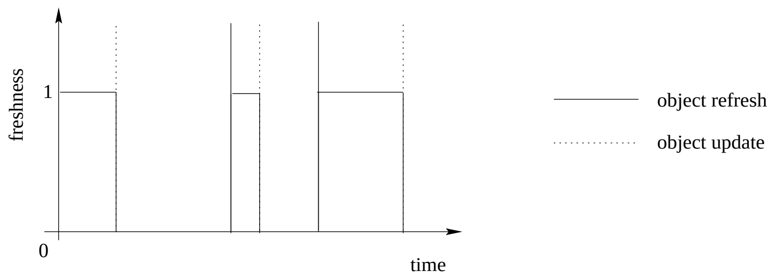


Fig. 2. An example of the evolution of the freshness of an object.

Thus, the freshness of an object O over a period $T = [t_i, t_j]$ is defined as:

$$F(O,T) = \frac{1}{t_j - t_i} \int_{t_i}^{t_j} F(O,t)dt \tag{1}$$

and the freshness $F(D,T)$ of a database consisting of N objects is defined as:

$$F(D,T) = \frac{1}{N} \sum_{i=1}^{N} F(O_i,T) \tag{2}$$

Using Equation 2, we must prioritize some object refreshes. This equation implies equal importance for all database objects. It is common though, that the more popular objects — the objects with higher access rates — contribute more to what is perceived as database freshness. This is due to the skewed access patterns for Web objects. Thus, we modify Equation 2 to take into account the access frequency f_O of an object and thus we have the following definition of the weighted database freshness:

$$F(D,T) = \frac{1}{N}\sum_{i=1}^{N} f_{O_i} * F(O_i,T) \tag{3}$$

The objective of our best-effort cache coherence maintenance scheme is to schedule object updates so as to minimize the database freshness as this is defined by Equation 3. Since the object update time is not apriori known, we deal with an on-line problem. We seek for an efficient heuristic, which may probably exploit some information about the popularity of the object, or some characteristics of the update stream.

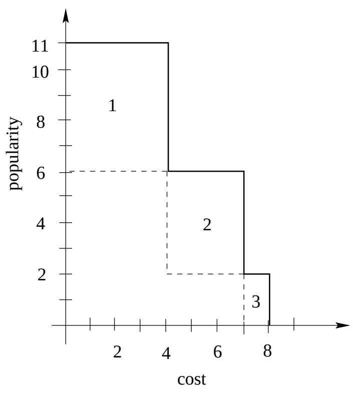
4 Scheduling Web View Updates

When there are sufficient network and processing resources, we can schedule an object refresh as soon as that object has undergone an update. This implements a simple FIFO scheduling policy. In our case though, the resources are limited and thus the server maintains a queue containing all the refreshes that must be executed. Nevertheless, we can still resort to the FIFO policy, but there is an open question of whether we can do better.

Let us visualize our scheduling problem using *2-dimensional Gantt charts*, which were introduced in [11]. Suppose that we have three pending refreshes in the server's queue. We will refer to these as Refresh1, Refresh2 and Refresh3 with total processing cost (measured in some cost unit) Cost1 (=4 units), Cost2 (=3 units) and Cost3 (=1 units), respectively. Let also the popularity of the respective objects (measured in some popularity unit) be Pop1 (=5 units), Pop2 (=4 units) and Pop3 (=2 units). Suppose that the three refreshes occurred with the order mentioned, i.e., first Refresh1 then Refresh2 and finally Refresh3. Then, FIFO would produce the following schedule $S_{FIFO} = \langle \text{Refresh1}, \text{Refresh2}, \text{Refresh3} \rangle$. The left part of Figure 3 illustrates the cost incurred by FIFO, which is the total area under the heavy solid line. This area is equal to 64. It is straightforward to deduce that this cost directly represents our divergence metric. Thus, to increase the freshness we must decrease the total area of the schedule.

As shown in [11], the minimum value for that area for non-preemptive schedules is obtained if we schedule the refreshes in non-increasing value of the slope $\rho_i = \frac{pop_i}{cost_i}$. Thus, we have $\rho_1 = \frac{5}{4} = 1.25$, $\rho_2 = \frac{4}{3} = 1.33$ and $\rho_3 = \frac{2}{1} = 2$. Under this rule, the *largest slope rule* as we will call it, we obtain the following schedule $S_{LSR} = \langle \text{Refresh3}, \text{Refresh2}, \text{Refresh1} \rangle$. The cost (area) of this schedule is equal to 58, leading to an improvement of 10% even for this small example.

Using the scheduling rule defined by the slope of each refresh, we initiate the transmission of object updates. This rule is no longer optimal if there are "dependencies" between the objects. A dependency between two objects o_1 and o_2 exists when o_2 is derived by o_1 and thus it makes no sense to refresh o_2 before refreshing o_1. Nevertheless, we use the *largest slope rule* to schedule refreshes, but paying attention to avoid unnecessary refreshes that may occur due to object derivation hierarchies.



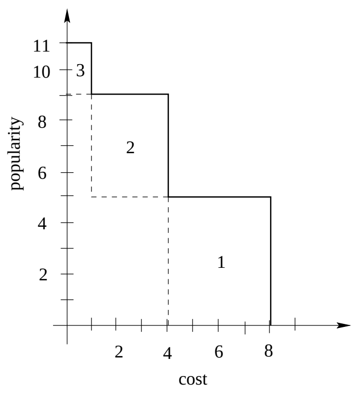


Fig. 3. Divergence for FIFO (left) and LSR (right).

5 Performance Evaluation

To investigate the performance of the proposed scheduling policy, we conducted a series of tests comparing it with the FIFO policy. We describe the simulated system model in section 5.1 and a subset of the results obtained in section 5.2.

5.1 System model

We used the PASASOL library[1] for building and simulating a distributed system environment. PARASOL gives the ability to define hardware and software components and simulates them.

Hardware Architecture The hardware architecture consists of k CDN servers, x routers/gateways and network links between the servers and the routers. One of the CDN servers is located in the same site (or just same subnetwork) with the back-end database and we call it the masterCDN. A graphical illustration of the hardware components is presented in Figure 4.

The routers play the role for routing the messages between the CDN servers based on the route tables built during the system initialization. We have chosen randomly real IP addresses for each of the CDN servers scattered all over the world (US, Australia, Europe etc) and we used the *traceroute* utility to find the real network connectivity.

All the PARASOL network links are set to have the same connection speed, thus the time for transferring a file from node A to node B, is up to how many routers/gateways are between node A and B. This is a plausible assumption and fits the reality quite well.

Every CDN server consists of one CPU except for the masterCDN node, which consists of tree CPUs. That's because we want to estimate separately the computation time used by the DBMS, which is a component independent from our scheduler. All the CPUs are set to the same CPU speed.

[1] http://www.cs.purdue.edu/research/PaCS/parasol.html

Our simulator is open, and everything is configurable. The default configuration variables we used for the hardware components are shown in Table 1.

Parameter	Default	Interpretation
links speed	2000 Kbps	Data Links speed
cpus speed	200 MIPS	CPUs speed

Table 1. Baseline system parameters related to hardware components.

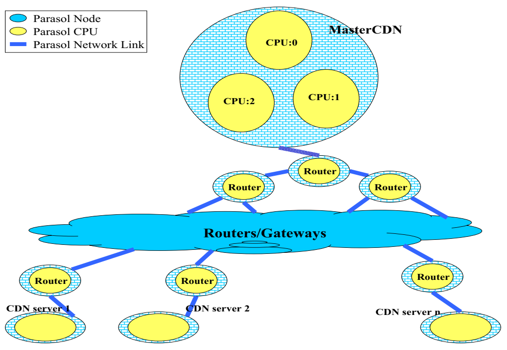


Fig. 4. Simulated System Hardware.

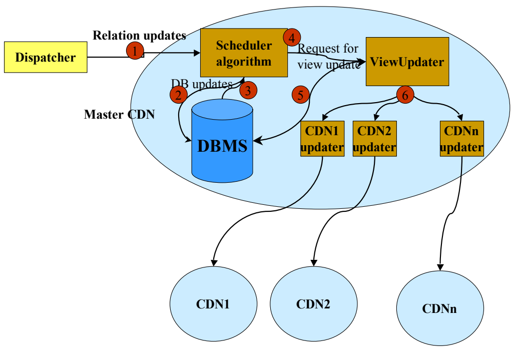


Fig. 5. Simulated system model.

Software Architecture/System Tasks Every node of our system runs specific tasks for which an overview is show in Figure 5.

Routers. On every router/gateway node, a router task is running. This task receives packets (files/webviews) from the network. Every packet consists of the header and the body (the file itself). The header contains the destination node. Upon receiving a packet from node A to node B, the router task sends it to the next node of the shortest path from A to B. This ”next node” may be another router node, or the node B itself if the specific router node is connected with a link with node B.

CDNservers. On every CDN server (except for the masterCDN) an ftp service is running. This service just receives files (webviews) from the network and sends back an ACK message to the masterCDN node.

MasterCDN. This is the most core node of the system and runs several tasks which are shown in Figure 6.

– **DBMS.** The DBMS runs on CPU 1 of the masterCDN node. It is responsible for receiving relation update requests from the Scheduler and executes them. The DBMS task processes the relation update requests sequentially. After completing

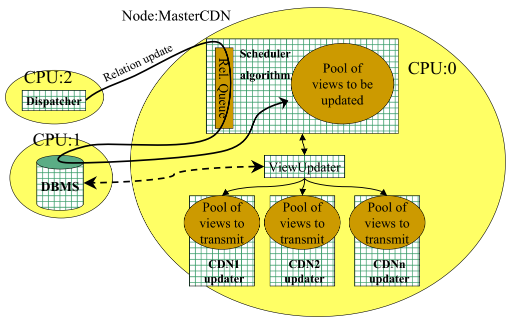


Fig. 6. masterCDN components.

a relation update it sends back to the Scheduler task an ACK message . This ACK message has the meaning that the specific relation update has finished. This helps us in two ways:

- The relation update requests are non-blocking, so our algorithm continues to run while a relation update is performed.
- The Scheduler has full control or the progress and will schedule a Web view update iff the relation update has been completed. So every Web view will be valid on its generation time, and it is guaranteed that no invalid data are used to compute the views.

– **Scheduler.** The scheduler task is running on CPU 2 of the masterCDN node. It is the core element of the system and handles almost everything. It maintains 2 queues in parallel. One for the relation update requests and one with the webviews that have to be updated. Upon receiving a relation update request, (a) it is added in the relations queue, (b) all the webviews depended on this relation are added in the webviews queue. Whenever the DBMS task is idle, the next relation update is sent to it (using always the FIFO). Whenever the ViewUpdater is idle, a webview is selected (based on the appropriate algorithm) and sent to the ViewUpdater.

– **ViewUpdater.** It is running on CPU 2 of the masterCDN node. Upon receiving a webview update request from the scheduler
  - it computes the view
  - notify all the CDNupdaters
  - sends back to the scheduler an ACK that the task has finished

– **CDNiUpdater.** On CPU 2 of the masterCDN node are running $k-1$ CDNUpdater processes. Each CDNUpdater corresponds to one CDN server and is responsible for transferring/sending the updated views to the corresponding CDN server. Each CDNUpdater task maintains its own pool of webviews that have to be sent to the appropriate CDN server. This way the transfer between masterCDN to CDNserver(x) is independent on the transfers to any other CDNserver. The selection of the webview that is to be sent, is made by the selected algorithm (FIFO or SMART).

– **Dispatcher.** Is running on CPU 0 of the masterCDN. It is responsible for producing relation updates. The update rate is configurable as shown in Table 2.

Parameter	Default	Interpretation
relation cost update	8000	Cost (in instructions) for a relation update
avg view update cost	1	View update cost (relative to relation cost)
Scheduler algo	FIFO, LSR	Algorithm for webview update selection
CDNupdater algo	FIFO, LSR	Algorithm for webview transmission selection
dispatcher rate	1.0	Dispatcher update rate
dispatcher rate relative to	DBMS	rate is relative to: DBMS,Updater,Net
Dispatcher rate deviation	0.0	Deviation for the dispatcher rate.

Table 2. Baseline system parameters related to software components.

Model Architecture We assume the existence of a backend database that contains M relations, which hold the source data. A number N of Web views are defined for our system. Some of these views are directly derived by the base relations, some others are derived by relations and other views and other are derived only by other views. Thus configuration subsumes the existence of an object derivation graph similar to that described in [14]. Updates take place only on the relations and are propagated in the views. Each edge server has a copy of every Web view. Thus, a copy of the i-th view $(1 \leq i \leq N)$ located at the j-th edge server $(1 \leq j \leq K)$ will be denoted as $v_{i,j}$. Each view $v_{i,j}$ is associated with a popularity factor $f_{i,j}$, in the range $[0, 1)$. Table 3 shows the value of the baseline parameters.

Parameter	Default	Interpretation
num_relations	100 or 500	Number of base relations
num_views	2000 or 6000	Number of views
view_size_min	2000 bytes	Minimum view size
view_size_max	5000 bytes	Maximum view size
view_size_avg	4000 bytes	Average view size

Table 3. Baseline system parameters.

We created a *view derivation graph* similar to that described in [14] to simulate the effect that some views may be generated by other views and not only from base relations. This directed acyclic graph has 3 levels. The lowest level is represented by the base relations, the next level is composed by 1000 views and the third level consisted of 1000 (in case for sparse graphs) or of 5000 views (in case of dense graphs). The dependencies of each view is measured by the average in-degree. The base relations participation is the view generation is uniform. The in-degree for the level-2 views is 2 for sparse graphs (4 for dense graphs) and the in-degree for the level-3 views is 4 for sparse graphs (6 for dense graphs).

The popularity of each view is determined by a uniform distribution. This setting (uniform popularities and uniform refresh costs) favors FIFO, which has no consideration about access rate or update cost. We decided though to examine the performance of the policies under this setting so as to validate the superiority of our method even in neutral (though unrealistic) environments.

5.2 Experiments

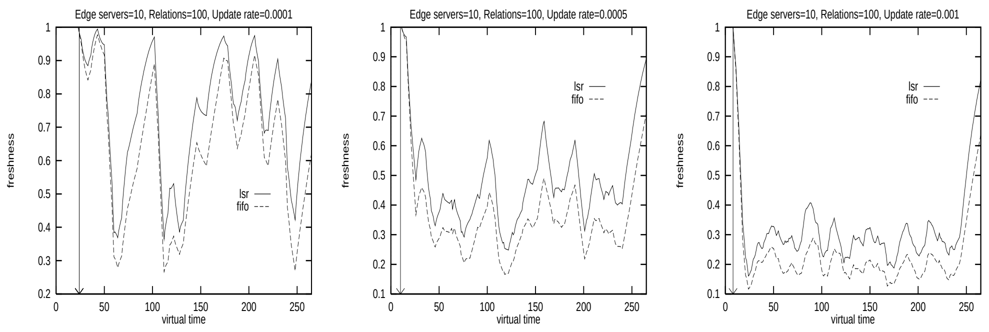


Fig. 7. Freshness vs update rate — (Left) 0.0001, (Middle) 0.0005 and (Right) 0.001.

Our first experiment investigates the performance of LSR and FIFO for various updates rates. The results are depicted in Figure 7. We can see that LSR significantly outperforms FIFO. The gains are always more than 15%.

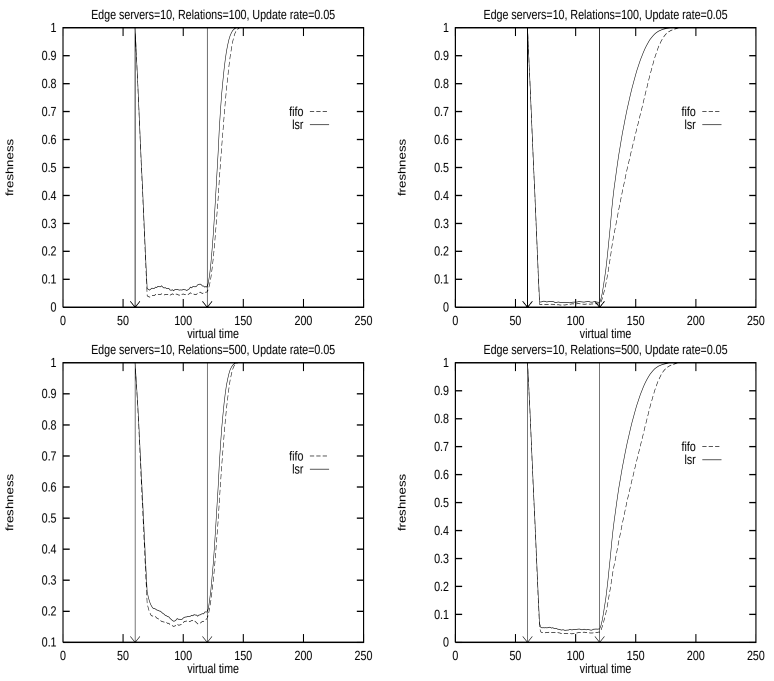


Fig. 8. Freshness — (Top) 100 relations, and (Down) 500 relations. (Left) Sparse dependency graph, and (Right) Dense dependency graph.

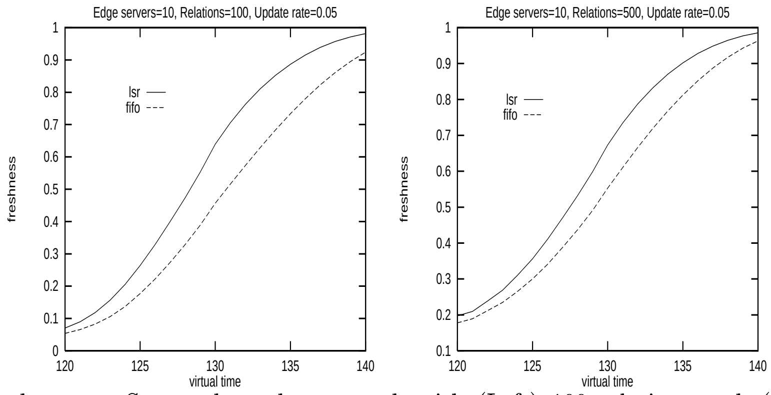


Fig. 9. Freshness — Sparse dependency graph with (Left) 100 relations and, (Right) 500 relations.

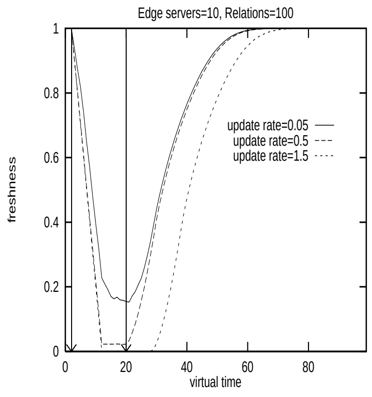


Fig. 10. Freshness achieved by LSR for various update rates.

Our second experiment examined the capability of the algorithms to recover after a surge of updates that force freshness to drop to very small values. The results are depicted in Figure 8. Again, we can see that LSR recovers much faster that FIFO. Figure 9 presents the same results, though it zooms into the interval which includes the time required by the algorithms to recover.

6 Conclusions

In this article, we proposed and experimentally verified a best-effort cache coherence maintenance scheme for the edge servers of a Content Delivery Network. We defined a coherence model based on a Boolean metric of the freshness of a cached object and then we proposed a policy to disseminate object updates to the edge servers. Our scheduling policy takes into consideration the access rate to the outsourced objects and the cost of updating an object in order to prioritize the refreshes. Using synthetically generated data, we presented experimental evidence that the proposed policy is significantly better than the FIFO scheduling method.

Finally, in Figure 10 we present the scalability of our algorithm to recover for various update rates. We observe that no matter how large the update rate is, LSR can recover very fast. This is illustrated by the steep slope of the curve.

As future work, we plan to investigate the alternative of organizing the CDNs into a (possibly deep) hierarchical structure, so as to parallelize the sending of the updated content and compare its design issues and performance with the present approach.

References

1. L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the Web. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, pages 550–561, 2002.
2. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, pages 562–573, 2002.
3. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven Web sites. In *Proceedings ACM International Conference on Management of Data (SIGMOD)*, Santa Barbara, CA, 2001.
4. J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic Web data. In *Proceedings IEEE International Conference on Computer Communications (INFOCOM)*, New York, NY, 1999.
5. J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings ACM International Conference on Management of Data (SIGMOD)*, pages 117–128, 2000.
6. E. Cohen and H. Kaplan. Refreshment policies for Web content caches. *Computer Networks*, 38(6):795–808, 2002.
7. P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P.J. Shenoy. Adaptive push-pull: Disseminating dynamic Web data. In *Proceedings International World Wide Web Conference (WWW)*, pages 265–274, 2001.
8. J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.
9. L. Do, P. Ram, and P. Drew. The need for distributed asynchronous transactions. In *Proceedings ACM International Conference on Management of Data (SIGMOD)*, pages 534–535, 1999.
10. A. Gal and J. Eckstein. Managing periodically updated data in relational databases: A stochastic modeling approach. *Journal of the ACM*, 48(6):1141–1183, 2001.
11. M.X. Goemans and D.P. Williamson. Two-dimensional Gantt charts and a scheduling algorithm of Lawler. *SIAM Journal on Discrete Mathematics*, 13(3):281–294, 2000.
12. D. Katsaros and Y. Manolopoulos. Cache management for Web-powered databases. In D. Taniar and W. Rahayou, editors, *Web-Powered Databases*, pages 201–242. IDEA, 2002.
13. A. Labrinidis and N. Roussopoulos. WebView materialization. In *Proceedings ACM International Conference on Management of Data (SIGMOD)*, pages 504–511, 2000.
14. A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the Web. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, 2001.

15. A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized Web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5), 2003.
16. A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative leases: Scalable consistency maintenance in Content Distribution Networks. In *Proceedings International World Wide Web Conference (WWW)*, pages 1–12, 2002.
17. C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, pages 144–155, 2000.
18. C. Olston and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings ACM International Conference on Management of Data (SIGMOD)*, pages 355–366, 2001.
19. C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proceedings ACM International Conference on Management of Data (SIGMOD)*, pages 73–84, 2002.
20. C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings ACM International Conference on Management of Data (SIGMOD)*, pages 377–386, 2001.
21. K. Ramamritham. Real-time databases. *Journal of Distributed and Parallel Databases*, 1(2):199–226, 1993.
22. J.W. Wang, D. Evans, and M. Kwok. On staleness and the delivery of Web pages. *Information Systems Frontiers*, 5(2):129–136, 2003.