階層式動態物件更新控制之研究
Hierarchical Consistency Control for Caching of Dynamic Contents

Jonathan C. Lu (呂俊賢) Huang-Chu Chen (陳皇助)
jonlu@csie.fju.edu.tw ivan0904@yahoo.com.tw
Department of Computer Science and Information Engineering
Fu Jen Catholic University, Taipei, Taiwan

Abstract

With the help of increasing bandwidth and advanced software technologies, dynamic web pages have been widely applied to various web sites. HTTP/1.1 has provided new control headers for caching the dynamic pages at the proxy efficiently, but it still cannot satisfy all the requirements for dynamic web caching. In this work we present a mechanism to maintain data consistency of various degrees by using active push. We developed an analytic formula for calculation of the update interval Δ given the tolerance value defined by the users depending on the type of object requested. We then designed a hierarchical structure where for each object requested a dissemination tree is created rooted at a leader proxy selected via a hash function. Object updates are pushed from the server to the leader, and further disseminated along the tree only when necessary. We evaluated our design by conducting a trace-based simulation, and the numerical results showed that our analytic formula agreed with the simulation very well. The results also displayed a twenty-eight percent saving in communication bandwidth for the hierarchical tree structure over the flat one.

Keyword: Consistency control, caching, dynamic objects, performance analysis

1. Introduction

Recent studies have shown that an increasing fraction of the data on the Web has become time-varying because they are often generated on-the-fly at the server using dynamic mechanisms. Unfortunately, those dynamic web pages cannot be cached easily at the proxy servers, which has limited the hit rate and presented a performance bottleneck.

To generate dynamic web pages, the server usually has to retrieve information from databases and perform additional computing and processing. By contrast, static web pages are often generated from reading local files only. We can see that the generation of dynamic web pages consumes much more resources, which results in heavier server load and longer response time. If dynamic web pages can be cached at the proxy servers, both the bandwidth requirement and the server's load can be decreased, which in turn reduces the latency for the client requests.

In caching web pages to improve the hit rate, one important task of the proxy server is to maintain data consistency between the cached copy and its original data on the server. Instead of requiring strong consistency, however, users may have different tolerances for data consistency, depending on the type of the web pages. For example, a user may be willing to access pictures posted several hours ago, but would not accept a stock quote unless it is up-to-date. However, the contents of web pages may change in different rates, and if we try to maintain data consistency by traditional client-pull methods, the server may not be able to set a single suitable TTL value for all the cached pages in order to meet various requirements from the clients.

In this work, we will focus on how to maintain data consistency of various degrees when caching dynamic web pages at proxies. The server and proxy cooperate by using active push instead of passive pull. The rest of this document is organized as follows: Chapter two describes the related work. In Chapter three, we will develop an analytical model that computes the time interval between consecutive data pushes. This can help reduce the use of communication bandwidth. We will also propose a hierarchical dynamic object replacement architecture to make proxy servers interwork with each other. This architecture will offload from the server and distribute among the proxies.

Chapter four presents the numerical results obtained from a trace-input simulation, and Chapter five concludes the paper.

2. Related Work

There have been some researches that provided solutions for the caching of dynamic web pages. A cooperative consistency solution along with a mechanism called clustered lease was presented in [2] to achieve data consistency and load balancing between the servers and proxies in a content distribution network (CDN). Server notifications were propagated to cluster of proxies in a scalable manner. A technique for disseminating dynamic data such as stock prices and real-time weather information was proposed in [3]. It is based on an optimized efficient dissemination tree to maintain the data consistency between server and proxy by pushing the data to appropriate intermediate nodes at the right time. A method was presented in [4] that combine the Push-based and Pull-based techniques for handling time-varying web data such as sports scores and stock prices. These methods tried to take advantage of the features of both approaches in order to better satisfy diverse temporal consistency requirements.

3. System Architecture

3.1 System Operations

We assume that the client is allowed to specify the degree of data consistency depending on the type of object instead of always requiring complete strong consistency. For each object the client needs to register a value of consistency tolerance ranging from zero to one. For example, a tolerance of 0.1 for an object means that the proxy is allowed to return an object that may be inconsistent with the origin server for no more than ten percent of the requests. Because different clients may have different consistency tolerances on the same object, the server does not need to push the new content to all the clients every time the data is updated. When the proxy registers the specified object at a server, one lease is granted. The lease denotes the interval of time during which the server agrees to notify the proxy when the specified object is modified. If the lease has expired and the proxy receives a new requirement, the proxy is required to obtain a new lease. Although the lease with long duration can make a server push the updated data to the proxy constantly and result in faster response to the clients, it can also possibly waste the bandwidth if the updated data pushed to the proxy has not been accessed by the clients very often. This work will not cover the problem of how to determine the optimal lease duration. Interested reader can refer to [7].

3.2 System Architecture Overview

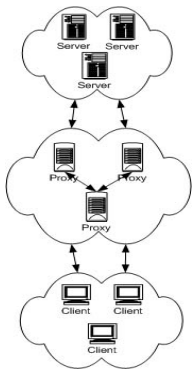

Figure 1. System architecture diagram.

In this subsection we provide an overview of own system architecture. Figure 1 shows a typical web access architecture with proxy servers. We assume that for each object requested the client needs to express its consistency requirement by registering an error tolerance value, ε, to the proxy server. For the same object, different clients may specify different tolerance value depending on their requirement. Upon receiving a request for an object currently not cached, the proxy forwards the request to the server. The server employs a Δ-Consistency mechanism, which means that it agrees to send to the proxy at most one update notification every Δ time units (no matter how many updates have happened) and no later than Δ time units after an update. The server then computes the update interval Δ according to the tolerance value sent by the proxy, and decides whether to accept the request based on the calculated Δ and its available resource. If yes, the server will notify the proxy of both the value of Δ and the length of the lease.

3.2.1 Calculation of Δ

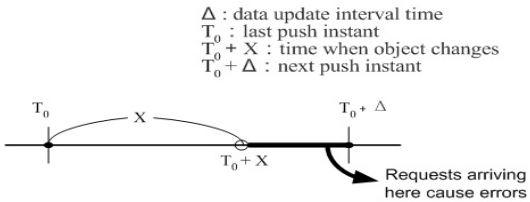

Figure 2. Data update interval.

We assume that requests for the same dynamic object arrive at the system following a Poisson process with an arrival rate λ. The intervals between consecutive changes of an object follow an exponential distribution with mean equal to $1/\mu$. Figure 2 displays a data update interval. Let

T_0 be the instant of last data push. Assume the object changes its values at $T_0 + X$. Since the new value is not pushed to the proxy until $T_0 + \Delta$, any request arriving at the proxy between the time interval ($T_0 + X$, $T_0 + \Delta$) will retrieve an incorrect result. Let P_k denote the probability that k requests arrive in such an interval and obtain erroneous results. We have

$$P_k = \int_0^{\Delta} \frac{[\lambda(\Delta - x)]^k}{k!} e^{-\lambda(\Delta - x)} \mu e^{-\mu x} dx \quad (1)$$

Let $\overline{E}$ be the average numbers of requests in error during an update interval. $\overline{E}$ can be computed as follows:

$$\begin{aligned} \overline{E} &= \sum_1^{\infty} kP_k = \int_0^{\Delta} \sum_1^{\infty} k \frac{[\lambda(\Delta - x)]^k}{k!} e^{-\lambda(\Delta - x)} \mu e^{-\mu x} dx \\ &= \int_0^{\Delta} \lambda(\Delta - x) \sum_1^{\infty} \frac{[\lambda(\Delta - x)]^{k-1}}{(k-1)!} e^{-\lambda(\Delta - x)} \mu e^{-\mu x} dx \\ &= \int_0^{\Delta} \lambda(\Delta - x) \mu e^{-\mu x} \quad dx \\ &= \lambda \left[\Delta - \frac{1}{\mu}\left(1 - e^{-\mu\Delta}\right) \right] \end{aligned} \quad (2)$$

The error rate is defined as the number of requests in error divided by the total number of requests. Therefore, we have

$$ErrorRate = \frac{\overline{E}}{\lambda\Delta} = 1 - \frac{1}{\mu\Delta}\left(1 - e^{-\mu\Delta}\right) \quad (3)$$

Given an error tolerance rate specified by the client, the server will use equation 3 to calculate the update interval length Δ. Note that when $\mu\Delta$ is a very small value, equation 3 can be further simplified as $1 - \frac{1}{\mu\Delta}\left(1 - \left(1 - \mu\Delta + \mu^2\Delta^2\right)\right) = \mu\Delta$. Thus, $\Delta = \frac{Error\ rate}{\mu}$.

3.3 Server System Modules

Figure 3 shows the modules that are added to the server.

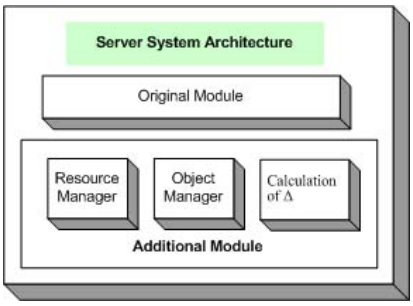

Figure 3. Server system architecture diagram.

3.3.1 Resource Manager

When a proxy sends an initial request or a request to modify the current tolerance value, the server first calculates the update interval length Δ, and then runs the Resource Manager to check if there is sufficient resource to satisfy the request. The resources may include CPU utilization, bandwidth of network connection, memory storage, etc. In this work, we only consider the bandwidth of network connection in the system. Since there will be at most one update sent out every Δ time units and the update happens once every $1/\mu$ time units on the average, for an object of size S the required bandwidth at the server is roughly equal to $S * \min\left(\frac{1}{\Delta}, \mu\right)$. The request will be rejected by the server if its available bandwidth is less than $S * \min\left(\frac{1}{\Delta}, \mu\right)$.

3.3.2 Object Manager

The function of the Object Manager is to compute the time of next push. In principle, there will be at most one push every Δ time units. However, if the object did not change its value within the last Δ time units, the push will be deferred to the instant when the object changes. Note that the push will be executed only if the lease for the object has not expired.

3.4 Proxy System Modules

The proxy includes four modules: Resource Manager, Object Manager, Leader Selection and Tree Manager. The Resource Manager and Object Manager are identical to those on the server. Figure 4 displays the structure of a proxy.

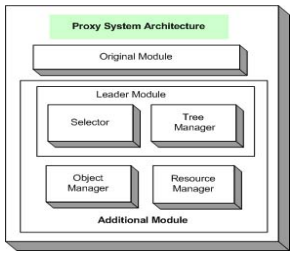

Figure 4. Proxy system architecture diagram.

3.4.1 Leader Selection

In order to reduce the load at the server, a leader selection algorithm is executed to select a leader proxy for each object. Upon receiving a request for an object, the proxy will forward the request to the selected leader. This leader is responsible for propagating update notifications to the other proxy servers. The leader gathers the requests from the proxy server group and presents only the smallest ε value to the server.

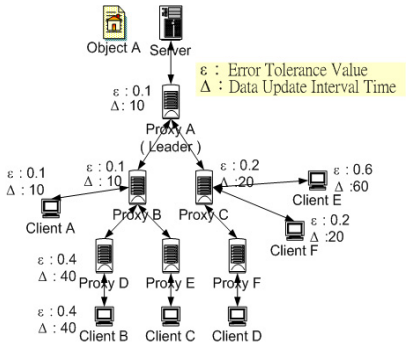

Figure 5. Singular object system architecture diagram.

Figure 5 illustrates the tree structure created to maintain the data consistency for a single object. The leader is responsible for communicating with the server and maintaining of the tree. A hierarchical tree is created for each object requested, thus there can be thousands of trees in the system. The proxy leader of an object can be determined by employing a hashing function such as the MD5 hash [8] of the object URL. An advantage of the hash-based approach is better load balancing among the proxies. Figure 6 depicts the sequence of leader selection.

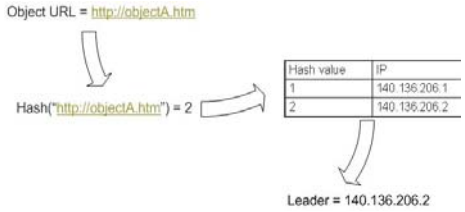

Figure 6. Leader selection procedure.

3.4.2 Tree Manager

Figure 5 show that clients may specify different tolerance values to a proxy for the same object. We use the minimum value to represent the error tolerance value at the proxy. When a client requests an object from the proxy, the proxy will send a request message to the leader. However, instead of pushing updates directly to all the proxies that it serves, it organizes the proxies as a tree structure where proxies with smaller Δ are placed closer to the root of the tree. Each node in the tree receives updates from its parent and propagates down to its children. For example, let Δ_P and Δ_Q (where $\Delta_P \leq \Delta_Q$) denote the Δ values of a proxy P and its child node Q in the tree, respectively. To meet node Q's requirement, proxy P only needs to forward one update to Q for every $[\Delta_Q/\Delta_P]$ updates it receives, where [x] denotes the largest integer less than or equal to x. There will be some bandwidth waste if the ratio Δ_Q/Δ_P is not exactly an integer, so we always try to insert a new proxy Q as a child of P where Δ_Q is as close to an integer multiple of Δ_P as possible.

The algorithm is as follows: When a proxy Q needs to be inserted, we first identify the nodes whose Δ value are less than Δ_Q. We then divide Δ_Q by each of these Δ values and record their remainder. The node with the minimal remainder will be designated as the parent. The shape of the tree will certainly affect the system performance. A wider tree means that each node has many children. Consequently, the parent node needs to send updates to many proxies (child nodes), which will increase the computing delay. On the other hand, a slender and deeper tree will increase the transmission delay for update notification to propagate to nodes at lower levels.

The maintenance of the tree includes Insert, Relocate and Delete operations, which are explained in the following:

- Insert: When a proxy receives a first request for an object from a client, it will send a registration message to the leader, who then inserts the proxy to a proper position in the tree. Figure 7 illustrates the sequence of proxy insertions into a tree. In step 1 there is an empty tree initially. In step 2 a node A with Δ = 20 is inserted under the leader directly since the tree is empty. Step 3 shows that another node B with Δ = 10 needs to be inserted. Because Δ_B is smaller than Δ_A, we place node B under the leader directly also, and change the leader's Δ to 10. Finally, a node C with Δ = 30 needs to be inserted. It is positioned as a child of B since Δ_C is an integer multiple of Δ_B. Step 4 displays the final tree.

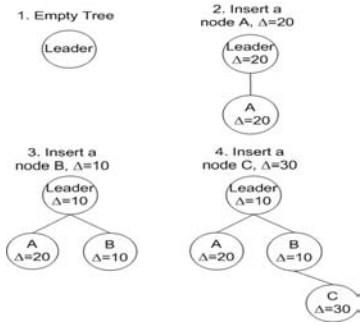

Figure 7. Example of node insertion.

- Relocate: When a proxy receives a request from a different client asking for an object that has been previously requested, it will first compare the new error tolerance value to the current one. If the new value is smaller (i.e., more stringent) than the current one, the proxy will send a request to the leader to have its update interval Δ

. shortened in order to meet the more stringent consistency requirement. Since the new Δ for the proxy becomes smaller, the proxy may no longer be in a proper position and the leader needs to rearrange the proxy and all its subtrees in the tree. Figure 8 illustrates this rearrangement. Step 1 shows the initial tree. Now assume that node D receives a move stringent request and its Δ is reduced from 30 to 25. Step 2 shows that D is rearranged as a child of A because Δ_D is now closest to a multiple of Δ_A. Step 3 and 4 display that the left and right subtrees of node D are rearranged to their proper places following the rule, respectively.

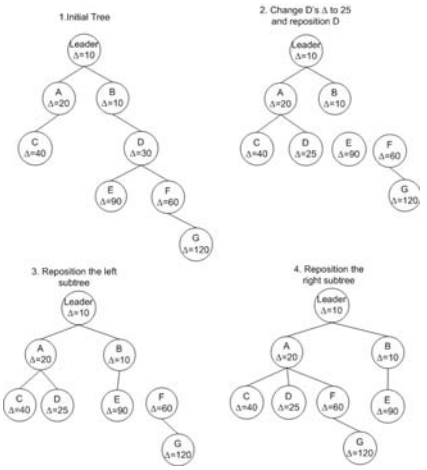

Figure 8. Example of node relocation.

- Delete: When the lease of an object is expired, the leader will notify all the proxies in the tree, and then delete the tree.

4. Simulation Results

Based on the proposed architecture, we have developed a simulator in Visual C++ to evaluate the performance of our design. The workload for our simulator is generated using the UC Berkeley Home IP Web Traces [9] which contains information that was gathered from November 1st to November 7th, 1996. The information includes client IP address, server IP address, request URL, the time of the request, the length of the response data, etc. The simulator runs on Pentium 4 1.6 GHz PCs with Microsoft Windows XP installed. The characteristics of the trace are shown in Table I and Table II.

Table I. Trace Characteristics.

Subject	Server	Client	Object	Request
Quantity	7094	6024	43381	1324870

Table II. Object Types.

Type	Object file extension	Total number of request	Average lifetime
0	gif , jpg	37827	16.8 hours
1	html , shtml	4637	5.6 hours
2	cgi , class , asp	1417	2.4 hours

We first verified the correctness of our analytic model for calculating Δ. Based on observations in [4] [6], we divide the objects into three types: Type 0 objects are primarily static images that change very infrequently, Type 1 objects consist of static texts that change slowly, and Type 2 objects are the components generated dynamically using techniques such as CGI, ASP or JSP that change their values frequently. The average lifetimes for Type 0, 1 and 2 objects are assumed to be 16.8, 5.6 and 2.4 hours, respectively.

Figure 9 shows the relationship between the tolerance ε and update interval Δ obtained from both analytic modeling and simulation. It shows that as the tolerance value increases (i.e., less stringent), the update interval becomes longer. Note that our analytical results agree with the simulation very well.

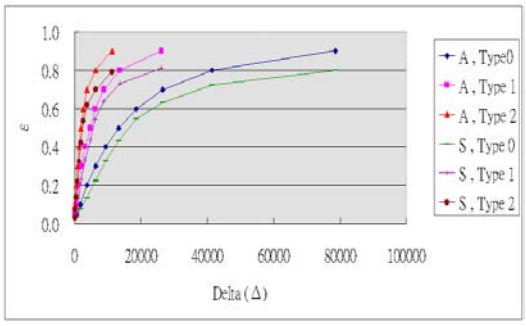

Figure 9. Relationship between ε and update interval Δ (A:anaytical, S:simulation).

The server will reject a request if it does not have sufficient resource available. We can see in Figure 10 that given fixed communication bandwidth at the server, the percentage of rejected request is very high when the client requirement is stringent (i.e. with small ε). The rejection rate drops as ε becomes large. We can also see that the rejection rate for the case of a longer lease is higher because each lease holds up the resource at the server much longer. The rejection of requests can be alleviated and even

completely eliminated if we equip the server with enough amount of bandwidth (=1.5Mbps in the simulation), which can be seen in Figure 11.

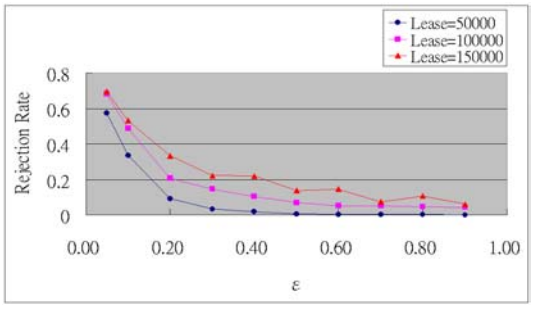

Figure 10. Rejection Rate (Bandwidth = 56Kbps).

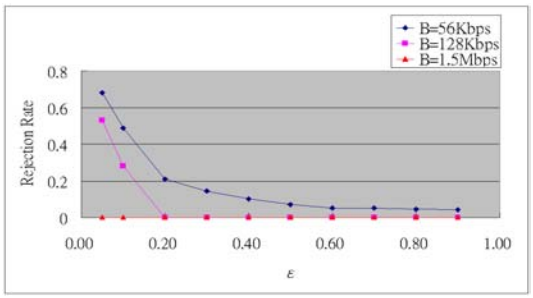

Figure 11. Rejection Rate (Lease = 100000s).

Control messages are exchanged between the server, the leader and the proxies when initial request for an object or requests to modify ε values are issued. Figure 12 and 13 plot the number of control messages exchanged in the system. When ε is small and many request are rejected, a lot of control message are exchanged because each rejected request would first contact the leader, which in turn contacts the server before being rejected. As ε becomes larger, more requests will be accepted. Once a request has been accepted, all the subsequent requests for the same object do not need to be forwarded to the server unless the associated ε is smaller. Therefore, the number of control message decreases.

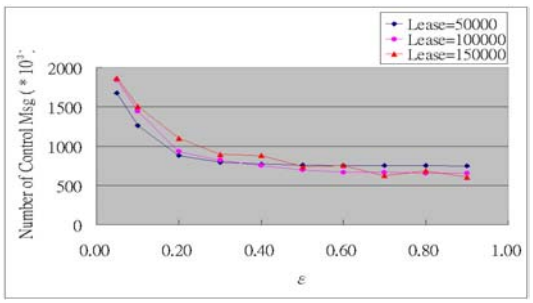

Figure 12. Number of Control Messages (Bandwidth = 56Kbps).

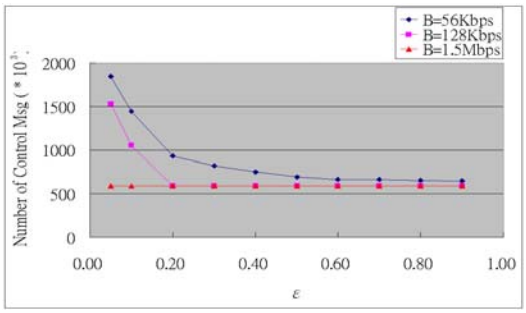

Figure 13. Number of Control Messages (Lease = 100000s).

To understand the benefit of using a hierarchical tree for disseminating updates, we measure the total number of push updates for both the case of hierarchical tree and the case of flat structure where all the proxies requesting the same object are positioned as direct children of the leader. In the latter case, the leader is assumed to broadcast the update to all its children as soon as it receives one from the server. As shown in Figure 14, there is little number of updates when ε is small since many requests have been rejected. As ε becomes large and more requests have been accepted, the number of updates increases. As ε increases further, however, the corresponding Δ becomes a large value and the server sends updates less often than before, which causes the number of updates to drop a little bit as can be seen from the figure. The saving in number of updates is about twenty-eight percent for using hierarchical tree over the flat structure. If the server is equipped with sufficiently large bandwidth (=1.5Mbps in the simulation), then the number of updates is at its peak when ε is small because all the requests are accepted, as shown in Figure 15. Again, the number of updates drops as ε increases.

Figure 16 and 17 display the total number of bytes pushed versus ε. Because most of the object size in the trace falls in the range of one to ten kilobytes, the results are similar to those in Figure 14 and 15.

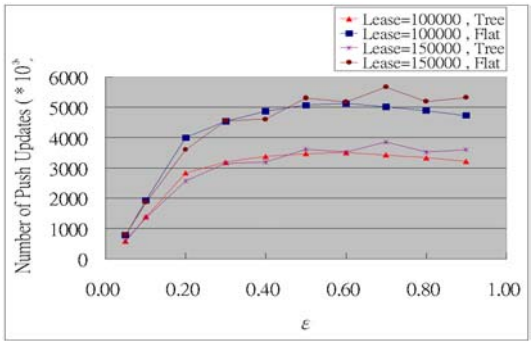

Figure 14. Number of Push Updates (Bandwidth = 56Kbps).

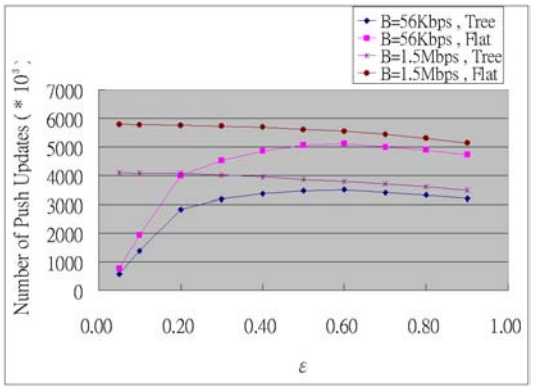

Figure 15. Number of Push Updates (Lease = 100000s).

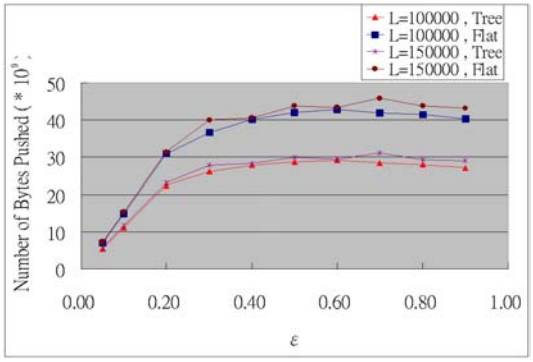

Figure 16. Number of Bytes Pushed ε (Bandwidth = 56Kbps).

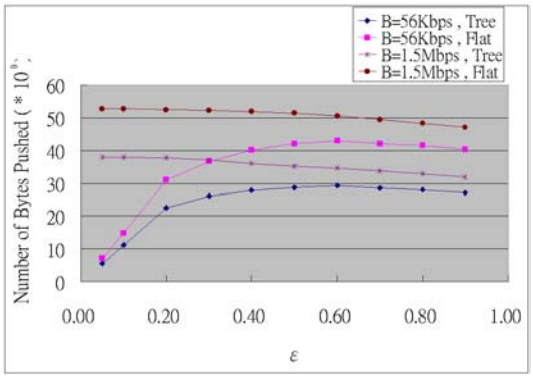

Figure 17. Number of Bytes Pushed (Lease = 100000s).

5. Conclusions

In this work we proposed a mechanism of Δ-consistency for the server and proxy that provide services for time varying web pages (such as stock quotes). We developed an analytic formula for calculation of the update interval Δ given the tolerance value defined by the users depending on the type of object requested. We then designed a hierarchical structure where for each object requested a dissemination tree is created rooted at a leader proxy selected via a hash function. Object updates are pushed from the server to the leader, and further disseminated along the tree only when necessary. We evaluated our design by conducting a trace-based simulation. Numerical results showed that our analytic formula agreed with the simulation very well. The result also displayed a twenty-eight percent saving in communication bandwidth for the hierarchical tree structure over the flat one.

Acknowledgment

This work was sponsored by the National Science Council of Taiwan R.O.C. under contract number NSC91-2745-P-030-004.

References

[1] D. Xu, C. Chun Tung, Y. Zongkai, C. Wenqing and H. Jiaqing, "A Self-Organizing Scheme for Cache Consistency", *Proceedings of the ICT 2003*, March 2003.

[2] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham and R. Tewari, "Scalable Consistency Maintenance for Content Distribution Networks", *Proceedings of the 11th World Wide Web Conference*, May 2002.

[3] S. Shah, K. Rammamritham and P. Shenoy, "Maintaining Coherency of Dynamic Data in Cooperating Repositories", *Proceedings of the 28th VLDB Conference*, 2002

[4] B. Krishnamurthy and J. Rexford, *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*, 2001, Addison Wesley Professional.

[5] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham and P. Shenoy, "Adaptive Push-Pull: Disseminating Dynamic Web Data", *Proceedings of the 10th World Wide Web Conference*, May 2001.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-lik Distributions: Evidence and Implications.", *Proceedings of INFOCOM*, March 1999.

[7] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Using Leases to Support Server-Driven Consistency in Large-Scale Systems", *in International Conference on Distributed Computing Systems*, 1998.

[8] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web", *Proceedings of the 29th ACM Symposium on Theory of Computing*, 1997.

[9] Steven D. Gribble, "UC Berkeley Home IP HTTP Traces", July 1997. Available at http://www.acm.org/sigcomm/ITA/.