Automated Delivery of Web Documents Through a Caching Infrastructure

Pablo Rodriguez Ernst W. Biersack Keith W. Ross
Institut EURECOM, FRANCE
{rodrigue, erbi, ross}@eurecom.fr

February 18, 2000

Abstract

The dramatic growth of the Internet and of the Web traffic calls for scalable solutions to accessing Web documents. To this purpose, various caching schemes have been proposed and caching has been widely deployed. Since most Web documents change very rarely, the issue of consistency, i.e. how to assure access to the most recent version of a Web document, has received not much attention. However, as the number of frequently changing documents and the number of users accessing these documents increases, it becomes mandatory to propose scalable techniques that assure consistency.

We look at one class of techniques that achieve consistency by performing *automated delivery* of Web documents. Among all schemes imaginable, automated delivery guarantees the lowest access latency for the clients. We compare pull- and push-based schemes for automated delivery and evaluate their performance analytically and via trace-driven simulation. We show that for both, pull- and push-based schemes, the use of a caching infrastructure is important to achieve scalability. For most documents in the Web, a pull distribution with a caching infrastructure can efficiently implement an automated delivery. However, when servers update their documents randomly and servers cannot ensure a minimum time-to-live interval during which documents remain unchanged, pull generates many requests to the origin server. For this case, we consider push-based schemes that use a caching infrastructure and we present a simple algorithm to determine which documents should be pushed given a limited available bandwidth.

1 Introduction

Due to the explosive growth of the World Wide Web, Internet Service Providers (ISPs) throughout the world as well as Application Service Providers (ASPs) are aggressively installing Web caches in order to reduce average client's latency, bandwidth usage, and server's load. Web caches keep copies of Web documents to satisfy future client requests from the cache without traversing peering networks or contacting the origin servers. However, placing geographically dispersed copies of a master document in Web caches creates the problem of *document consistency*, that is, documents delivered to the client from the cache should be fresh and consistent with the master copy at the origin server. Historically, Web caches have been designed to take arbitrary decisions on content freshness by using some time-to-live heuristics [9] [16]. However, using time-to-live heuristics, caches sometimes provide stale documents to the clients, which is unacceptable for many content providers and clients. Given the increasing number of Web sites that offer highly time-dependent information, e.g. news, stocks, weather maps, *strong consistency*, that is, always providing the freshest document to the clients, is becoming more and more important.

There are three main mechanisms for providing strong consistency in Web caches i) *validation*, ii) *invalidation*, and iii) *replication*. With validation, for every client request the cache checks with the origin server if the document copy is fresh. If the document copy is still fresh, the cache will answer the client with its cached copy. If the document has expired, the client will receive the master document from the origin server and the cache will keep a document copy. With invalidation, caches do not take any decision about the freshness of the document. Instead, when a document expires the origin server sends an invalidation message to those caches that store the expired document. Using invalidation, the first client request after the document was invalidated experiences high latencies since the document needs to be retrieved from the origin server. With replication, every document update is pushed into the caches, thus preloading the cache with always fresh information and freeing the cache from making any freshness decision. Using replication, clients always experience very small latencies, however, the bandwidth wasted can be considerable if no client requests the replicated information.

The choice between validation, invalidation, or replication should be made dynamically on a per-document basis and can be tailored to optimize different parameters, i.e., bandwidth usage or client latency. For instance, to optimize bandwidth usage in the ISP, the decision of which mechanism to use for strong consistency should be based on how frequently a document changes and how frequently it is requested. It is not the goal of this paper to analyze and develop a mechanism that dynamically selects the best scheme to provide strong consistency on a per-document basis, which is the topic of [14]. Instead, in this paper we look only at replication, one of the three techniques to keep strong consistency in Web caches. With replication, as opposed to invalidation or validation, client's latency is always minimized since documents are always delivered from the cache. In addition, replicating the actual document update removes the need for sending invalidation messages, or if-modified-since requests.

One way to implement replication is using an *automated delivery*. With automated delivery, caches subscribe to a document and new document updates are distributed to the caches without the clients making explicit requests. Documents are always fresh in the local cache, thus, when the client requests a document it is immediately delivered with minimal delay. Automated delivery can be implemented with a client-initiated *automated pull* mechanism or with a source-initiated *true push* mechanism. Automated pull, works as follows: Clients automatically poll the origin server periodically. If the origin server has updated its document, a client's poll will fetch the updated version. If the origin server has not updated its document, a client's poll will result in a not-modified reply. With true push, on the other hand, client-caches do not need to periodically poll the origin server. Origin servers send the latest document update to the subscribed clients as soon as the document update occurs. The primary reason to use pull, is that it uses the existing HTTP protocol [7] and that it does not require the origin servers to know where to push a document and to keep state information about the subscribed clients. Additionally, clients can asynchronously poll for new updates and they do not require to be synchronized to receive a new update as it is the case for push. A drawback of automated pull, however, is that origin servers may receive an excessive number of polling messages. unicast from the origin server to the clients.

In this paper we develop simple and rigorous analysis for pull- and push- based automated delivery, and study the impact of a caching infrastructure in both, pull and push. In a caching infrastructure [5] , caches are placed at different levels of the Internet (e.g., institutional, regional, and national). Caches act as application-layer multicast nodes providing an *asynchronous reliable multicasting infrastructure* [13]. We will see that for pull- and push-based schemes, a caching infrastructure is very important to achieve scalability. More precisely, we find that pull with a caching infrastructure can effectively implement an automated delivery for most of the documents in the Web, while providing strong consistency. When i) a document is updated at fixed times, or ii) when a document is updated randomly and servers specify a small time-to-live interval during which a document remains unchanged, automated pull with a caching infrastructure uses bandwidth very efficiently, and sends few requests to the origin servers. Using analytical models and trace-driven simulations we quantify the minimum time-to-live interval that servers should set to their documents for a caching infrastructure to filter out most of the poll-requests to the origin server. We also quantify the number of cache tiers needed for a caching infrastructure to mimic a multicast distribution.

In the rare case that servers can not ensure even a small time-to-live interval during which a document is not updated, implementing strong consistency with an automated pull generates many poll requests to the origin server and does not scale. In this latter case, a push distribution should be implemented. In this paper we consider a pure push distribution which uses a caching infrastructure. Given the large number of documents that can be potentially pushed and the limited bandwidth available for pushing, there are only certain documents that can be pushed. Documents pushed, experience small latencies since they are delivered from the local cache, documents not pushed experience high latencies since they are delivered from the origin server. Therefore, there is a need for an intelligent algorithm that determines which documents should be pushed into the caches to provide the minimum average latency to the clients and fully utilized the available bandwidth. We present a simple mechanism to determine which documents should be pushed in a limited bandwidth environment and study its impact with a trace-driven simulation.

The rest of the paper is organized as follows. In Section 2 we study automated pull with a caching infrastructure and we analyze its bandwidth usage. In Section 2.2 we calculate the total polling rate at the origin server with and without a caching infrastructure. We divide the total polling rate into requests that result in a document not-modified and requests that result in a new document transfer. Section 3 describes a push scheme through a caching infrastructure. Section 3.1 presents and evaluates an algorithm to decide which documents to push in a bandwidth limited environment.

2 Automated Pull with Caching Infrastructure

In this section we analyze automated pull to provide strong consistency, with and without a caching infrastructure. We consider bandwidth usage (Section 2.1), and polls at the origin server (Section 2.2, 2.3, and 2.4). We do not consider latency since documents are updated in the background (i.e. latency is zero).

2.1 Bandwidth Analysis

In this section we first obtain simple expressions for the *bandwidth usage* as a function of client interest in a document for i) pull without caching infrastructure (unicast), ii) pull with caching infrastructure, and iii) end-to-end multicast. Then, we study the bandwidth usage in a more general scenario with random networks and non-uniform demand distribution. We define the bandwidth usage as the total number of links traversed to distribute a document update from the origin server to the clients.

To obtain simple analytical expressions for the bandwidth usage we model the Internet as a full O-ary tree with L tiers. Each tier has a depth of H links, therefore, the total height of the tree is LH links. At the root of the tree there is the origin server, and clients are at the leaf nodes of the tree. Let l be the level of the network tree, where level $l = 0$ indicates the leaf nodes of the tree, $0 \leq l \leq LH$. To model a multicast distribution we assume that at every node of the network tree there is a multicast router. To model a caching architecture, we suppose that at the root of every tier there is a cache, except at the root of the top tier, where there is the origin server. We refer to the caches in the leafs as *leaf caches*. We refer to the caches that are the closest to the origin server as *top-level caches*. There are O^H *top-level* caches, and O^{LH} leaf caches.

Let p be the probability that a leaf cache requires a certain document. We assume that the total demand for a document is uniformly distributed among all leaf caches. Using a uniformly distributed document demand we try not to be biased in favor of multicast or a caching infrastructure; if the demand were concentrated around some nodes, the benefits of multicast or a caching infrastructure would be much higher due to the high sharing.

In the case of **unicast** the expected bandwidth usage, $BW_u(p)$, is the number of links between the origin server and an the leaf caches, LH, times the expected number of leaf caches interested in a document, $O^{LH}p$, i.e.,

$$BW_u(p) = LHO^{LH}p \ . \tag{1}$$

For the case when all leaf caches require the document update, $p = 1$, the bandwidth needed is $BW_u(1) = LHO^{LH}$.

When pull uses a **caching infrastructure**, document requests can be satisfied from intermediate caching levels, saving bandwidth in the upper network levels. The expected bandwidth usage of pull with a caching infrastructure, $BW_h(p)$, is the number of links H between every cache level times the expected number of caches at a given level that require the document update. A cache at level l requires the document update with a probability $1 - (1 - p)^{O^l}$, which is the probability that at least one of the O^l leaf caches rooted at level l is interested in that document. Thus,

$$BW_h(p) = \sum_{l \in \{0, H, 2H, \ldots, (L-1)H\}} O^{LH-l} H (1 - (1 - p)^{O^l}). \tag{2}$$

When all leaf caches are subscribed to a document $BW_h(1) = H\frac{O^{(L+1)H} - O^H}{O^H - 1} \simeq HO^{LH}$, for $O^H >> 1$.

In the case of **multicast**, packets can be replicated at every network level and share all network links from the origin server to the leaf caches. The expected bandwidth usage $BW_m(p)$ is the sum of the expected number of links traversed at every network level. The number of links traversed between level $l + 1$ and level l is equal to $O^{LH-l}(1 - (1 - p)^{O^l})$, where $1 - (1 - p)^{O^l}$ is the probability that the multicast tree is extended to level l. Thus,

$$BW_m(p) = \sum_{l=0}^{l=LH-1} O^{LH-l}(1 - (1 - p)^{O^l}) \ . \tag{3}$$

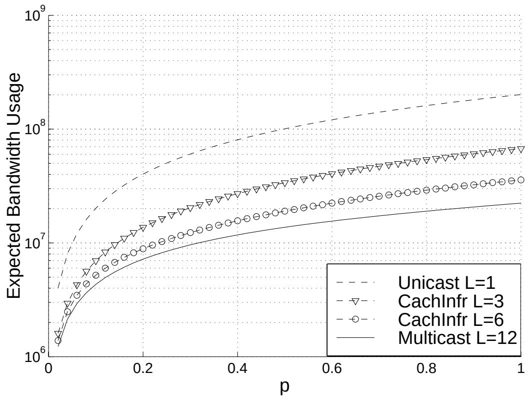
When all leaf caches require the document update, $p = 1$, we have $BW_m(1) = \frac{O^{LH+1} - O}{O - 1} \simeq O^{LH}$, for $O >> 1$.

Table 1 summarizes the bandwidth usage of unicast, multicast, and a caching infrastructure for $p = 1$ and $O >> 1$. We observe that a caching infrastructure reduces L times the bandwidth usage compared to unicast. Thus, using a caching infrastructure is attractive when there are many cache tiers L. Multicast uses H times less bandwidth than a caching infrastructure and LH times less bandwidth than unicast. Thus, when there are few cache tiers and the height of each tier is large, multicast uses significantly less bandwidth than caching, however, when there is a large number of cache tiers, a caching infrastructure uses similar bandwidth than multicast.

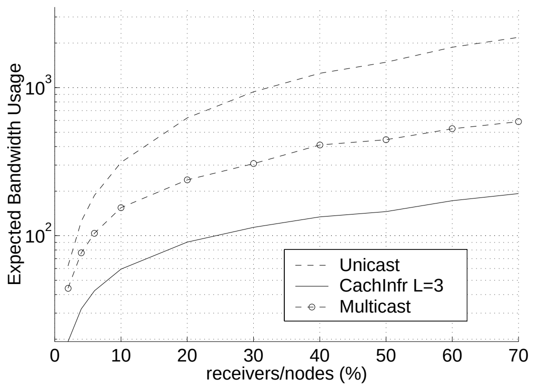
Table 1: Bandwidth usage for unicast, caching and multicast distribution. $p = 1$, $O >> 1$.

	Unicast	Caching Infrastructure	Multicast
Bandwidth	LHO^{LH}	HO^{LH}	O^{LH}

In Figure 1 we plot the bandwidth usage of the various schemes as p varies. For $L = 1$ there is no caching infrastructure. In this situation, the server and the leaf caches communicate via unicast. Comparing unicast and multicast, we see that the difference in bandwidth usage is rather pronounced once a document becomes moderately popular. For very popular documents, $p = 1$, unicast uses approximately $LH = 12$ times more bandwidth than multicast. For a caching infrastructure we have varied the number of cache tiers L from 3 to 6. As the number of tiers L increases, the bandwidth usage of a caching infrastructure gets



(a) Full O-ary tree and uniform document demand. Total height of the tree $LH = 12$ links. $O = 4$.



(b) Random network topology and non-uniform document demand.

Figure 1: Bandwidth usage for unicast, multicast and a caching infrastructure.

closer to the bandwidth usage of multicast. Given that the total height of the tree $LH = 12$ links, a caching infrastructure with 12 cache tiers ($L = 12, H = 1$) perfectly mimics a multicast distribution in terms of bandwidth, however, this would mean that there is a cache at every multicast router. For a reasonable number of cache tiers (i.e., $L = 4$), a caching infrastructure considerably reduces the bandwidth usage compared to no caching infrastructure, and has a bandwidth usage close to that of multicast, since it performs an application-level multicast distribution.

In Figure 1(b) we consider the case of a random network topology and non-uniform document demand. To simulate this scenario we used *tiers* [6], which is a random topology generator. We created a hierarchical topology of three levels: WAN, MAN, and LAN that aim to model the structure of the Internet topology. A WAN connects 20 MANs, and each MAN connects 9 LANs, resulting in a core topology of 225 nodes. To simulate a three level caching infrastructure we placed Web caches at the border nodes between the LANs and MANs, and the MANs and the WAN. We defined the bandwidth usage as the number of links traversed to deliver one packet from the source to the receivers. We calculated the bandwidth used by unicast, multicast with shortest path tree, and a caching infrastructure. Receivers are placed at random locations. From Figure 1(b) we observe that the results obtained with the random topology generator and a non-uniform demand distribution are very similar to those obtained with our analytical model (Figure 1(a)).

2.2 Total Polling Rate at the Origin Server

Now, we study the **total polling rate** at the origin server for automated pull to achieve strong consistency. We consider the cases when i) there are only leaf caches and there are no intermediate caches, and when ii) there is a full caching infrastructure.

With pull, clients periodically poll the origin server and fetch a new updated document version if the document has changed. If the times at which different clients poll servers to check for document updates are more-or-less synchronized, an origin server can receive an excessive number of polling messages over a short period of time. In order to alleviate this problem, the latest releases of client browsers delay the polling times by random times, thereby spreading out the polling messages [11]. To account for this random delay, we suppose that requests from multiple clients for the same document are Poisson distributed [8] (in Section 2.4 we also consider other probability distributions for the document requests). For each leaf cache we denote W as the interarrival time between document requests, which is exponentially distributed with an average request rate λ, $F(w) = 1 - e^{-\lambda w}$ (see Figure 2).

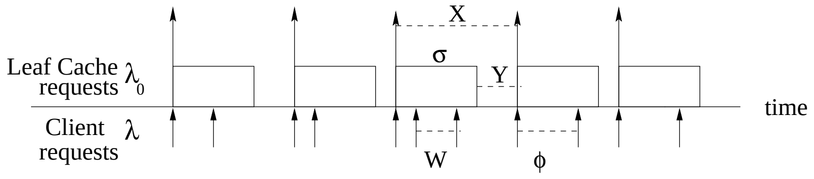


Figure 2: Polling rate at a leaf cache for a given max-age interval.

To model document's consistency we assume that origin servers specify a time-to-live value, σ, (using the max-age HTTP/1.1 cache-control header [7]) during which servers ensure that a document will remain unchanged. When a new document update is fetched from the origin server by any cache, the document is assigned an *age*, ϕ. The age of a document is the period of time that a document copy has been in any cache of the caching infrastructure, that is, the time since the document was retrieved from the origin server. A cache can serve a document with age ϕ if the max-age value, σ, is greater than ϕ (Figure 2). If origin servers wish to validate every request, they may set a max-age equal to zero, i.e. $\sigma = 0$, so that the caches always validate the document.

Some of the requests at the leaf cache, λ, will be filtered out by the leaf cache. Therefore, the rate λ_0 at which polling messages leave a local ISP, is less than λ. Referring to Figure 2, we now proceed to calculate λ_0. Since the time between requests W, is exponentially distributed and thus memoryless, the time Y between the end of a max-age interval and next poll request is also exponentially distributed with the same average request rate λ. Let X be the interarrival time between filtered poll requests. X is a random variable that is equal to the sum of the max-age interval and the random variable Y, that is, $X = \sigma + Y$. The expected value of X is $E[X] = \sigma + E[Y] = \sigma + \frac{1}{\lambda}$. Thus the expected poll rate λ_0 from a leaf cache is given by

$$\lambda_0 = \frac{1}{E[X]} = \frac{\lambda}{1 + \lambda \cdot \sigma} \quad . \tag{4}$$

Figure 3 shows the request rate leaving a leaf cache λ_0 for different max-age intervals, σ. We see that if the origin server wants to validate all requests (i.e., $\sigma = 0$), the polling rate λ_0 is equal to the client request rate λ. However, as soon as a small max-age is set (i.e., $\sigma = 5$ minutes), the polling rate from a leaf cache is kept low even for very high client request rates.

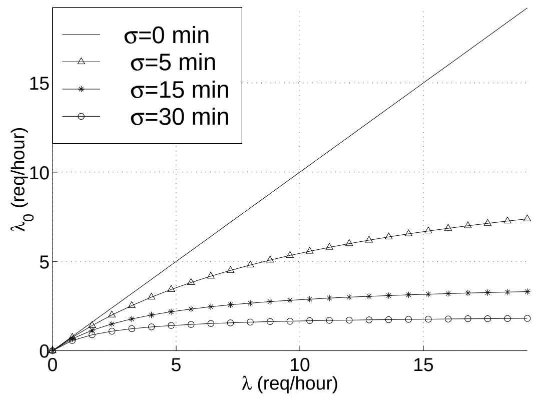


Figure 3: Polling rate λ_0 from a subscribed leaf cache depending on the clients request rate λ for different max-age intervals.

Next, we consider the case where there is a caching infrastructure. A caching infrastructure not only acts as an application-level multicast for distributing document updates (Section 2.1), but can also filter out many if-modified-since polls to the origin server. To illustrate this, in Figure 4 we show the case of a two-tier caching infrastructure (two leaf caches and one regional cache). When leaf caches poll the origin server, only the first poll from any leaf cache is redirected by the regional cache to the origin server. Further polls from any leaf cache during a max-age interval are filtered at the regional level (see Figure 4).

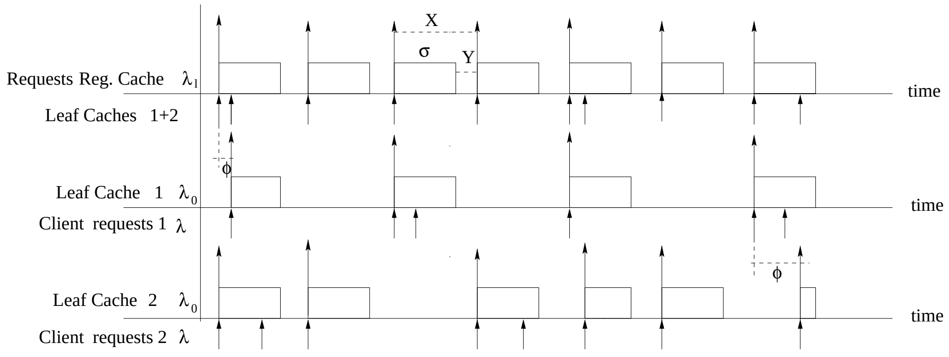


Figure 4: Request rate of a regional cache with two children leaf-caches

Let C_p be the number of leaf caches that are below one top-level cache, and let C_t be the number of top-level caches. At a top-level cache, the interarrival time between polls to the origin server is given by $X_t = \sigma + Y_t$, where Y_t is exponentially distributed with an average request rate equal to λC_p. Thus, the polling rate generated by one top-level caches to the origin server is $\lambda_t = \frac{\lambda C_p}{1+\lambda C_p \sigma}$. The total polling rate λ_s^h at the origin server generated by the C_t top-level caches is given by

$$\lambda_s^h = \lambda_t C_t = \frac{\lambda C_p C_t}{1 + \lambda C_p \sigma} \quad . \tag{5}$$

For very popular documents ($\lambda \to \infty$), the total polling rate at the origin server with no caching infrastructure approaches $\frac{1}{\sigma} C_p C_t$ and the polling rate at the origin server with a caching infrastructure approaches $\frac{1}{\sigma} C_t$. Thus a caching infrastructure reduces the polling rate at popular servers by a factor of C_p, the number of leaf caches below one top-level cache. To prefetch every document update in the local caches with an automated pull, caches need to automatically poll the server at a rate $\lambda \geq \frac{1}{\sigma}$. For $\sigma = 0$, the polling rate from local caches tends to infinity and also a caching infrastructure does not filter any request, thus, resulting in a high burden for the origin server. When $\lambda \geq \frac{1}{\sigma}$, clients receive every document update with an average delay smaller or equal to $\frac{\sigma}{2}$. For $\lambda \geq \frac{1}{\sigma}$ clients experience a relax form of strong consistency, since clients receive every document update but with a small delay. However, this form of strong consistency may be well accepted by most content providers and clients.

Next, we quantify the max-age value σ that servers must specify for an automated pull with a caching infrastructure to generate only few requests to the origin server. Figure 5 shows the total polling rate at the origin server for different values of max-age values when a caching infrastructure is in place and when there is no caching infrastructure.

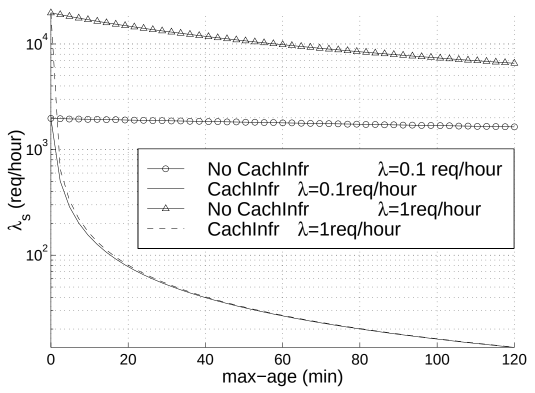


Figure 5: Total polling rate at the origin server for different max-age values and different client polling rates λ. $C_p = 728$, $C_t = 27$. Three-level caching infrastructure.

When there is no caching infrastructure the polling rate at the server stays high even for max-age values greater than zero. However, when there is a caching infrastructure, as soon as the server sets a small max-age time-to-live value during which the document is not updated (i.e., few minutes), a caching infrastructure reduces the number of requests to the origin server by several orders of magnitude. Setting a small max-age of few minutes can be tolerated by most Web sites, including those offering stock tickers, news, or weather maps which may not be updated faster than once every few minutes due to the way the information is generated.

In figure 5 we have also varied the client's polling rate λ. We see that as λ increases, the total polling rate at the origin server increases significantly when there is no caching infrastructure. However, when there is a caching infrastructure, the total polling rate at the origin server is kept low even for high client polling rates λ. We finally notice that the benefits of a caching infrastructure are less pronounced for small λ since there is a lower aggregation of requests at every parent cache. The higher the number of caches connected to a certain parent cache, the higher is the aggregation of requests at the parent cache, and therefore, the higher is the effect of a caching infrastructure in reducing the number of requests at the origin server.

2.3 Document Transfer Rate and Not-Modified Requests

In this section we divide the total polling rate at the origin server into polls that result in a not-modified response and polls that result in a document transfer.

Assuming that a document is updated every period t, the first poll in an update period results in the transfer of a new document update, HTTP code 200 [7] (see Figure 6). Then, the document is kept at the leaf cache during a max-age interval, σ, and all client requests during this interval are satisfied directly from the leaf cache. The subsequent client request after a max-age forces the leaf cache to send a poll request towards the origin server. If the document has not been updated, the poll results in a not-modified response, HTTP code 304 [7], and no document is transferred. If the request comes after a period t, the document is already expired and a new document update is transfered, code 200.

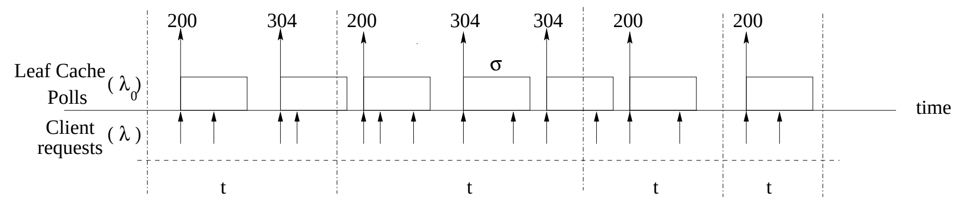


Figure 6: Distribution of document transfers (200) and requests for not-modified documents (304).

Next, we simulate a scenario where documents are updated randomly with a uniform distribution $P(t)$ in the interval $[T_{min}, T_{max}]$, e.g. a newspaper that publishes new information on its Web site at random intervals greater than $T_{min} = 10$ minutes, and that always updates its Web site every $T_{max} = 1$ hour. In Figure 7 we show the expected poll rate for not-modified documents and the expected document's transfer rate. We consider no caching infrastructure and a caching infrastructure. We have varied max-age from 0 to 2 hours. First we observe that for a max-age equals to zero, the number of requests for not-modified documents

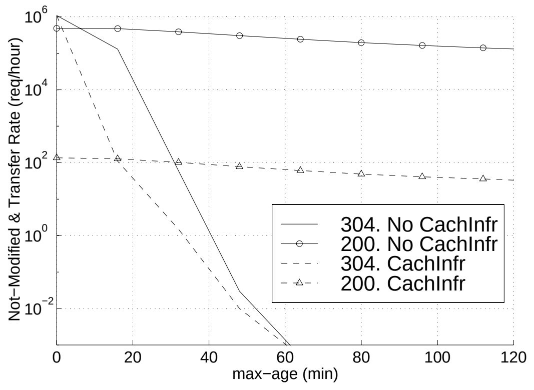


Figure 7: Polling rate for not-modified documents (304) and document transfer rate (200) for a caching infrastructure and for no caching infrastructure. $T_{min} = 10$ minutes, $T_{max} = 1$ hour. $\lambda = 10$ req/hour. $C_p = 4096$, $C_t = 64$. Three-level caching infrastructure.

(304) is very high with a caching infrastructure. However, as soon as a small time-to-live is allowed, i.e. max-age equals few minutes, the request rate for not-modified documents (304) decreases very fast, and this decrease is even steeper when there is a caching infrastructure since intermediate caches filter out most of the if-modified-since requests. For max-age greater than $T_{max} = 1$ hour, the number of requests resulting in not-modified documents is equal to zero since the every request finds a new document update.

The document transfer rate (200) is decreased by four orders of magnitude with a caching infrastructure, even for max-age equals to zero. For max-age values between 0 and 20 minutes the transfer rate is kept constant, which means that clients receive every document update with a maximum delay equal to max-age. For larger max-age values, the number of document transfers decreases, meaning that clients are obtaining sometimes a stale copy and do not get every document update. Therefore, setting a small max-age equal to few minutes, a caching infrastructure can efficiently reduce the number of requests for not-modified documents at the origin server (304), and the number of document transfers (200), while providing up-to-date information to the clients with small delays.

2.4 Trace Driven Simulation

The Poisson request rate and the network topology used so far are not intended to be an accurate model of the real poll rate and network topology. In this section we present trace-driven simulations with real workloads and a real caching infrastructure to confirm our analytical model. For this purpose we have taken 10 days of logs during November 1999, from the four major parent caches, “bo”, “pb”, “sd”, and “uc” in the National Web Cache infrastructure by National Lab of Applied Network Research (NLANR) [2]. These caches are the top-level caches of the NLANR caching infrastructure and they all cooperate to share their load.

To analyze the impact of a caching infrastructure on the polling rate of a popular origin server, we choose the CNN server which is one of the most popular Web sites [2]. CNN Web server provides news at any time of the day with no fixed schedule. We have considered the CNN welcoming page (http://www.cnn.com/, http://cnn.com) and we determined the total polling rate for this page for different max-age values, with and without the caching infrastructure. To obtain the topology of the caching infrastructure, we identified the different caches at the leaf or intermediate levels connected to the NLANR top-level caches that make a request for the CNN page. To simulate different max-age values, we have supposed that the CNN server sets a max-age value which varies from zero to two hours.

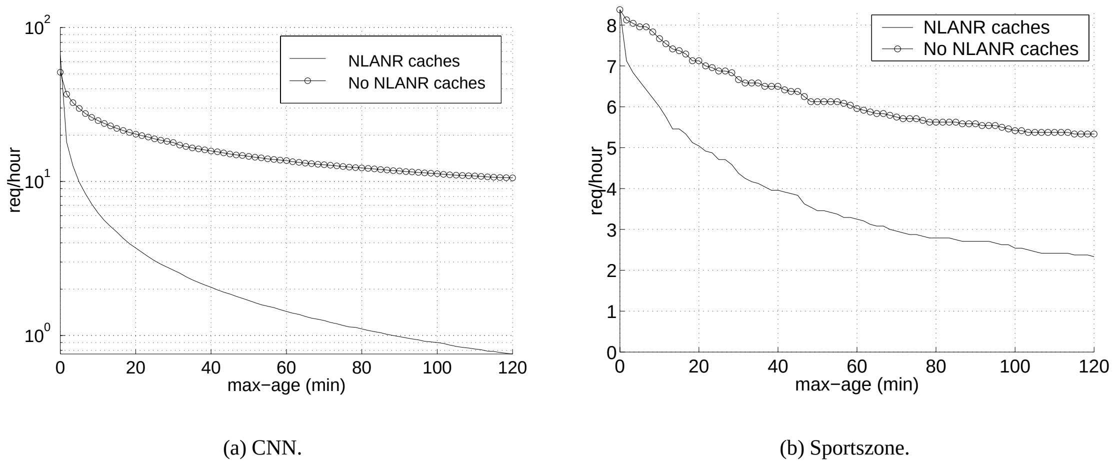


Figure 8: Trace-driven analysis. Poll rate at the origin server with and without NLANR top-level caches for different max-age values.

In Figure 8(a), we see the trace-driven simulation results for the CNN server, which are very similar to those ones obtained with our analytical model (Figure 5). When max-age is equal to zero, all polls go to the origin server regardless of the existence of a caching infrastructure. However, when a caching infrastructure is in place and the origin server sets a small time-to-live, i.e. max-age of few minutes, many if-modified-since requests are filtered at intermediate caches, cutting down on the number of polls to the CNN origin server by more than one order of magnitude.

In Figure 8(b) we have also analyzed an automated pull for the case of a less popular document. For this purpose we considered 10 days of logs from the welcoming page of the Sportszone Microsoft channel (http://channel-espn.sportszone.com/). As already showed in our analysis (Figure 5), the benefits offered by a caching infrastructure are smaller for unpopular documents than for popular documents. However, even for less popular documents the savings of hierarchical caching are still considerable, i.e., for the Sportszone page the NLANR caching infrastructure offers a reduction on the number of polls to the origin server of a factor of 2 or more as soon as a max-age of few minutes is set (Figure 8(b)) . To better validate our model we have also simulated many other (than CNN and Sportszone) Web sites and caching hierarchies. The results of these other simulations also match our analytical model very well.

Of course, a caching infrastructure requires cache servers to be purchased – perhaps a large number for each cache tier in order to have sufficient storage, processing power, and access bandwidth. But many Application Service Providers are prepared to pay the cost, as they are currently aggressively deploying caches in the networks [1]. In fact, it is quite common to find in today's Internet leaf caches with capacities in the order of GB and top-level caches in the order of several tens of GB [2] [15] since the price of the disks is dropping very fast [4] (for a more detailed analysis of the disk space see [12]).

3 Pushing with Caching Infrastructure

As we have discussed in the previous section, in the cases that i) documents are updated at fixed intervals, or ii) documents are updated randomly and origin servers set a small max-age interval, pull through a caching infrastructure uses bandwidth efficiently and significantly limits update checks sent to the origin server as well as document transfers. However, when documents are updated randomly and origin servers can not set even a small time interval during which documents remain unchanged, automated pull generates many requests to the origin server and does not scale. In this latter case, a push distribution scheme should be implemented.

If unicast is used to push document updates from the origin server to a large number of clients, two main problems arise: i) the origin server needs to keep a list with all the subscribed clients, ii) bandwidth is wasted because the same document copy is transmitted multiple times through the same links. To scale a push distribution from the origin server to the clients, a caching infrastructure can be used. A *Hierarchical Caching Push (HCP)* scheme can work as follows. Leaf caches subscribe to those documents that need to be pushed/replicated and the subscription is propagated up in the caching infrastructure to the origin server. All intermediate caches in the path from the client to the origin server also get subscribed to the document. A subscribed cache at level l, records the document's URL and the addresses of those caches at level $l-1$ that are also subscribed to the document. Origin servers keep the addresses of the top-level caches that are subscribed to their documents. When origin servers update a document, they push the document update to the subscribed top-level caches using unicast or multicast if available [12]. Top-level caches that receive a document update but that do not have subscribed children caches do not forward the update to the next tier. Top-level caches with subscribed children caches push the document update to the subscribed caches in the next tier. This process is repeated at every level of the caching infrastructure, until the document update gets to the subscribed leaf caches.

Subscriptions in the server/caches expire after a certain *lease time* and would need to be periodically refreshed by the clients [10] [17]. A hierarchical caching push also requires mechanisms for establishing and maintaining the caching infrastructure, as discussed in [18] and [14].

Intermediate caches can purge the document update once it has already been pushed to the leaf caches, thus, reducing disk space requirements. However, intermediate caches could also keep the document updates to satisfy requests from clients that are not subscribed to this document but still want to hit the document at a nearby cache.

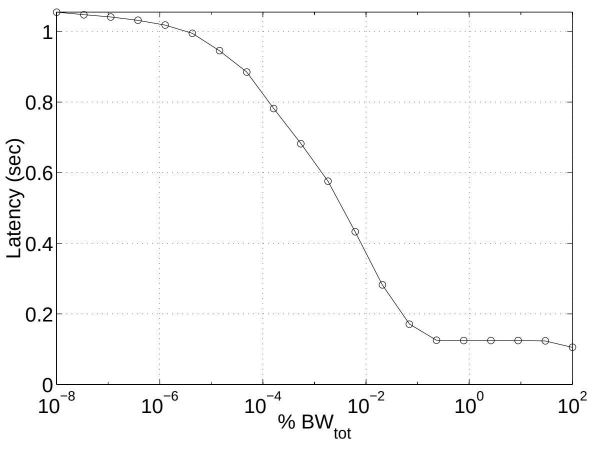
A hierarchical caching push, requires state information and processing at the caches/origin servers to know which are the subscribed caches at the next cache tier and push every update to the next-tier caches. However, the per-document state information and processing requirements can be greatly reduced by increasing the number of cache-tiers [12]. Even though the per-document state information and processing requirements can be very small by increasing the number of cache-tiers, caches need to keep state information for *every* subscribed document. If we assume that there are 1 million documents that require an automated delivery, and that for every page 58 bytes of state information are required (which is enough to store a URL, and keep track of 64 children-caches), top-level caches need to keep at most 55 MB of state information. This value is a very reasonable value given the current storage capacities of Web caches [15]. Also, from the total number of documents that require an automated delivery only few of them must be delivered with a pure-push distribution (see Section 2). Assigning a certain time-to-live to every subscription, i.e. a lease time [10] [17], caches can reduce even their storage and processing requirements, since only popular documents will frequently refresh their subscription. In addition, when there are multiple top-level caches, every top-level cache can take care of a different portion of the documents, thereby, sharing the load [14]. Documents can be group into categories (i.e., CNN sports) and caches only need to keep an entry for every category and not for every document. Finally, if multicast is available, the state information on the caches, and the processing requirements can be reduced even more [12].

3.1 What documents to push?

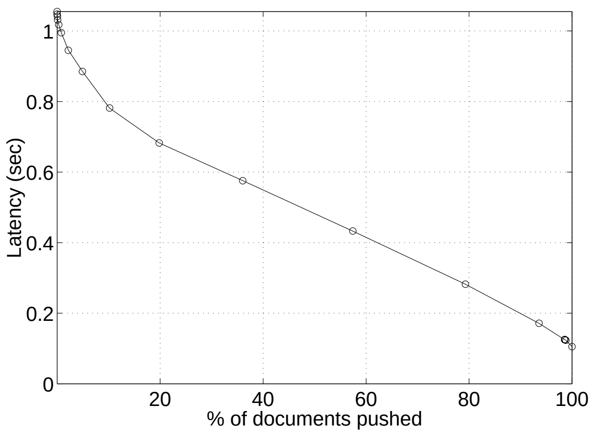
Up to now we have only considered the case of a single document and assume that there is infinite bandwidth for replication. However, caches usually need to decide which documents to push/replicate given that the available bandwidth for replication is limited. In this situation, an intelligent scheme must be used to decide which documents to push given a certain available bandwidth. The decision of pushing a document should maximize the number of total requests satisfied from the cache before a new document version needs to be pushed in the cache. To do this, the pushing algorithm should select for replication those documents that have a high number of requests per update period, and require little bandwidth consumption to keep these documents up-to-date. Note, that this algorithm minimizes the overall average latency experienced by the clients given a certain available bandwidth.

The pushing algorithm could be as follow. Every leaf cache determines the update period of a document t and its request frequency λ using the access logs (see [14] for more details). Given t, and λ, the leaf cache can calculate the number of requests per update period of a document as λt, as well as the bandwidth required to keep this document up-to-date as $BW = \frac{S}{t}$. Then, for every document, the leaf cache calculates the number of requests satisfied per unit of bandwidth consumed $\frac{\lambda t}{BW}$, and ranks all the documents based on this value, such that the first document is the one with the highest value. The leaf cache subscribes for replication on those documents that are ranked higher, until the number of selected documents is such that the bandwidth needed to distribute the selected documents is equal to the available bandwidth for replication. The documents being replicated will be delivered very fast from the leaf caches. The rest of the documents not replicated will be transfered from the origin server at a slower rate.

Next, we present the result of a trace-driven simulation that studies on a real environment how this pushing algorithm would work. We considered 10 days of logs from an ISP in the USA (AYE [3]), collected in Dec 1998. The total number of requests in the trace is roughly 10 million and there are about 400,000 different objects. From the logs we extract all the cacheable requests that contain last-modified information. We then extract objects of type text/html and image/gif. For every single object in the log-file we estimate the average request rate and the average update period. To calculate the average update period we use the average time difference between every request for the same object and the last-modified-time [14]. Then, we ranked all documents using the number of requests satisfied per bandwidth unit $\frac{\lambda t}{BW}$. We consider a situation where all clients are connected to a single leaf cache and the cache has a certain bandwidth available for replication. To simulate a real scenario, we used typical transmission rates of proxy caches [15].



(a) Latency versus % of bandwidth available



(b) Latency versus % of objects pushed

Figure 9: Expected client latency for different available bandwidth and different number of documents pushed. Assume that documents are ranked using the number of requests per bandwidth unit $\frac{\lambda t}{BW}$ and that higher ranked objects are selected for replication first. $BW_{tot} = 3.3$ Mbps

In Figure 9(a) we calculated the average latency experienced by a client when the bandwidth available for replication is equal to a certain percentage of the total bandwidth BW_{tot} Mbps, needed to replicate all objects. We see, that in the case that the available bandwidth is equal the total bandwidth BW_{tot} the latency experienced by the clients is very small and equals the latency experienced by a client when all documents are delivered from the cache. As the available bandwidth decreases, the latency experienced by the clients increases. However, the average latency experienced by the clients is the smallest possible latency given the current bandwidth. Any other scheme that would select a different set of objects to be pushed, would incur in a higher latency with the same bandwidth usage.

It is interesting to notice that it is enough for a cache to have a bandwidth equal to the 10% of the total bandwidth BW_{tot} to be able to provide very small client latencies. This is due to fact that there are a few very large documents that require a lot of bandwidth. However, these documents do not receive many requests and, therefore, it is not worth replicating them since they will not significantly reduce the average latency experienced by a client. In Figure 9(b) we have also plotted the average latency experienced by a client depending on the percentage of ranked objects replicated. We see that the experienced latency decreases very fast by replicating the high ranked objects. In fact, replicating the first 35% of the objects decreases the average experienced latency by almost one half. For low ranked objects, we see that the experienced latency decreases almost linearly with the number of objects replicated, and it is necessary to replicate about 95% of all objects to obtain similar latencies to those provided by a leaf cache. Even though this may seem a very large number, we should notice that these documents account for a very small percentage of the total bandwidth as shown in Figure 9(a). Also, we should note that in our simulations we did not modify the transmission rates of the origin servers when more and more objects are satisfied from the leaf cache. As more objects are delivered from the leaf caches, the transmission rates from the origin servers would increase, since the network and the servers are less utilized. As a result, the average latency experienced by a client would decrease at a higher rate than the one showed in Figure 9, requiring to push even less documents to obtain the same latencies.

4 Conclusions

In this paper we have proposed and compared different mechanisms to implement an automated delivery of document updates in the Internet to provide strong consistency. First, we have showed that an automated pull through a caching infrastructure can efficiently distribute all documents that are either (i) updated at fixed times or (ii) that are updated at random times but for which servers can set a small time-to-live interval during which documents do not change. Automated pull through caching infrastructure keeps the load (number of polls and document transfers) at the origin servers low and reduces the traffic generated by the poll messages. In addition automated pull with hierarchical caching uses bandwidth very efficiently, approaching the performance of multicast when there are many cache tiers.

In the rare situation where servers can not specify even a small time interval during which documents remain unchanged, automated pull generates too many polls at the origin server. For this later case, we have proposed a new scheme that pushes documents through a caching infrastructure. With a hierarchical caching push (HCP), caches act as application-level routers and filters forwarding the document only to those caches with clients subscribed. Finally, we have presented a pushing algorithm that determines which documents should be pushed to minimize the expected clients' latency given a certain available bandwidth and we have studied this algorithm using trace-driven simulations.

References

[1] "FreeFlow: How it Works. Akamai, Cambridge, MA, USA. Nov 1999".

[2] "National Lab of Applied Network Research (NLANR)", http://ircache.nlanr.net/.

[3] AYE, "http://www.aye.net/".

[4] E. A. Brewer, P. Gauthier, and D. McEvoy, "The Long-Term Viability of Large-Scale Caching", In *3rd International WWW Caching Workshop*, Manchester, UK, June 1998.

[5] A. Chankhunthod et al., "A Hierarchical Internet Object Cache", In *Proc. 1996 USENIX Technical Conference*, San Diego, CA, January 1996.

[6] M. B. Doar, "A Better Model for Generating Test Networks", In *Proceedings of IEEE Global Internet*, London, UK, November 1996, IEEE.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, et al., "RFC 2068: Hypertext Transfer Protocol — HTTP/1.1", January 1997.

[8] S. Gribble and E. Brewer, "System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace", In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[9] J. Gwertzman, "Autonomous Replication in Wide-Area Internetworks", M.S. Thesis, Harvard, Cambridge, MA, April 1995.

[10] C. Liu and P. Cao, "Maintaining Strong Cache Consistency in the World-Wide Web", In *Proceedings of ICDCS*, May 1997.

[11] Microsoft, "Webcasting in Microsoft Internet Explorer 4.0. White Paper", , September 1997.

[12] P. Rodriguez, E. W. Biersack, and K. W. Ross, "Automated Delivery of Web Documents Through a Caching Infrastructure", Technical Report, EURECOM, June 1999.

[13] P. Rodriguez, K. W. Ross, and E. W. Biersack, "Distributing Frequently-Changing Documents in the Web: Multicasting or Hierarchical Caching", *Computer Networks and ISDN Systems. Selected Papers of the 3rd International Caching Workshop*, pp. 2223–2245, 1998.

[14] P. Rodriguez and S. Sibal, "SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution", To appear in Proc. of the World Wide Web Conference 2000, EURECOM, AT&T Research Labs, Nov 1999.

[15] A. Rousskov, "On Performance of Caching Proxies", In *ACM SIGMETRICS*, Madison, USA, September 1998.

[16] D. Wessels, "Squid Internet Object Cache: http://www.nlanr.net/Squid/", 1996.

[17] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Using Leases to Support Server-Driven Consistency in Large-Scale Systems", In *18th International Conference on Distributed Computing System*, May 1998.

[18] H. Yu, L. Breslau, and S. Shenker, "A Scalable Web Cache Consistency Architecture", In *Proceedings of ACM SIGCOMM'99*, Cambridge, sep 1999.