SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution

Pablo Rodriguez [*]
Institut Eurecom
rodrigue@eurecom.fr

Sandeep Sibal
AT&T Labs Research
sibal@research.att.com

Abstract

We introduce SPREAD - a new architecture for distributing and maintaining up-to-date Web content that simultaneously employs three different mechanisms: client validation, server invalidation, and replication. Proxies within SPREAD self-configure themselves to form scalable distribution hierarchies that connect the origin servers of content providers to clients. Each proxy autonomously decides on the best mechanism based on the object's popularity and modification rates. Requests and subscriptions propagate from edge proxies to the origin server through a chain of intermediate proxies. Invalidations and replications travel in the opposite direction. SPREAD's network of proxies automatically reconfigure when proxies go down or come up, or when new ones are added. The ability to spontaneously form hierarchies is based on a modified Transparent Proxying mechanism, called *Translucent* Proxying, that sanitizes Transparent Proxying. It allows proxies to be placed in an ad-hoc fashion anywhere in the network - not just at *focal* points within the network that are guaranteed to see *all* the packets of a TCP connection. In this paper we (1) describe the architecture of SPREAD, (2) discuss how proxies determine which mechanism to use based on local observations, and (3) use a trace-driven simulation to test SPREAD's behavior in a realistic setting

Keywords: WWW, Scalable Content Distribution, Consistency

1 Introduction

Due to the explosive growth of the World Wide Web, Internet Service Providers (ISPs) throughout the world are installing Proxy caches to reduce user perceived latency as well as bandwidth consumption. Such Proxy caches are under the control of the ISP, and usually cache content for its client community, irrespective of the origin server. These Proxy caches are often called *Forward* Proxy caches to distinguish them from *Reverse* Proxy caches, which we discuss next.

More recently, several vendors, such as Akamai [1] and Sandpiper [17] have begun offering Proxy-based solutions to Content Providers, as opposed to ISPs. The business model here is that improving a user's browsing experience, is not only in the ISP's interest, but in the Content Provider's interest as well. This is becoming increasingly important, as the number of Content Providers multiply and compete for the attention of end users. Proxy caches used in such a scenario are often called *Reverse* Proxy caches, to underline the fact that they are controlled by and represent the interests of the Content Provider (or its agent). Reverse Proxy caches serve content on behalf of the Content Provider, usually to any arbitrary client on the Internet.

In the rest of the paper we use the term Proxy, Cache, and Proxy Cache interchangeably, since the Proxying and Caching functions are co-located in a single entity.

SPREAD can be realized in both the forward proxying and reverse proxying contexts. In this paper we consider the forward proxying context. Applying SPREAD in a reverse proxying context would need minor alterations,

[*]During the period of this work, he was at AT&T research labs as an intern.

which we point out at various points in the paper.

1.1 Object Consistency

One of the tenets of SPREAD is that the system provides *strong* object consistency. This means that content served is always fresh. Technically it is impossible to guarantee *absolute* freshness, since there is a non-zero delay between the time a proxy cache receives an object from an origin server, and the instant it serves it to a client. The term *strong* is used to distinguish it from *weak* schemes, which improve consistency but do not provide guarantees. We believe strong consistency is imperative, especially now that people have begun to rely on the Web in timely information for conducting business, and because an increasing number of sites have begun to offer time-sensitive information.

Forward proxies have been known to be notoriously sloppy in this area. While mechanisms exist within the HTTP protocol for maintaining cache consistency, in practice, forward Proxy caches administered by ISPs, use their own time-to-live (TTL) heuristics [8], [22] that are engineered in a rather arbitrary fashion. Historically, part of the effect (or some say the cause) has been that Content Providers often misuse or abuse features of the HTTP protocol, using techniques such as *cache-busting*. Regardless of how one sees this tension between ISPs and Content Providers, we believe that adhering to strong consistency mechanisms in accordance with the HTTP guidelines is important. This is a fundamental design guideline in SPREAD.

SPREAD uses three primary mechanisms to achieve strong consistency:

- Client Validation *(V)*: In this mechanism, for every client request that a proxy receives the proxy always checks back with the origin server to see if the object copy is fresh. This is typically accomplished by an *If-Modified-Since (IMS)* HTTP Request. If the origin server finds the object in the proxy fresh, the proxy cache will respond to the client with its cached copy. If the object has expired, the client will receive the master object from the origin server and the cache will keep an object copy. The only exception to this rule is if the object has been explicitly marked as cacheable, and a Max-age, Expires, or an equivalent piece of metadata has been set to a value by the origin server that indicates a non-zero time-to-live (TTL). In such a situation the proxy will not need check back with the server to validate the cached copy of the object for the stipulated TTL.

- Server Invalidation *(I)*: With invalidation, a proxy cache first subscribes to an invalidation service for that object (or range of objects) with the origin server, or an agent for the origin server that is responsible for signaling the expire of the object. In this case, the proxy cache assumes that the object is fresh unless an invalidation message from the origin server is received by the proxy to explicitly expire the object. Using invalidation, the first client request after the object is invalidated experiences high latency since the object needs to be retrieved from the origin server.

- Replication *(R)*: With replication, updated versions of the object are explicitly pre-loaded in the proxy cache by using *push*, or equivalently a *pseudo-push* that can be implemented with a periodic-pull. As with invalidation, a proxy cache must express interest in the object (or a range of objects) a-priori, by subscribing to the replication service. Using replication, clients always experience very small latencies, however, the bandwidth consumed can be wasteful in cases where there are more updates than requests.

To save on bandwidth, instead of sending the entire object, one may send just the *diff*, or some encoded form of the *change* between the old and new versions. This may be applied to all of the above mechanisms. The results of this paper remain valid under such a scenario as well.

1.2 Our Approach

A novel feature of SPREAD is that its proxies *dynamically* choose between client invalidation, server invalidation, and replication, on a per-object basis. This is discussed in detail in Section 4. Earlier studies [23], [9] have analyzed the benefits of server invalidation versus client validation, but their comparisons were in a context where strong cache consistency was not imperative. More importantly, their evaluation had been focused on assessing stale hit-rate using trace-driven simulations at a *macroscopic* level. In our work, we evaluate the competing mechanisms to keep strong consistency from a more fundamental perspective, analyzing the problem at the level of individual reads and writes of each object, which we believe yields substantial insight. The authors of [13] propose unicast invalidations instead of adaptive time-to-live mechanisms to keep strong consistency, however, using unicast communication from the server to the clients makes their approach non-scalable. The authors of [24] study the efficacy of server invalidations using a scalable distribution infrastructure, and provide several insights into the general problem of cache consistency. Our work advances the state-of-the-art beyond [24] in three major respects. First, proxies in our system *dynamically* choose the consistency mechanism based on their own observation of the request rates and update rates of objects. Prior knowledge of these statistics is not assumed. Second, our analytic results, help us define the thresholds of the optimal control policy at which proxy caches switch from one mechanism to another, which in turn helps us in building a smarter overall system. A third novel feature of SPREAD is its ability to spontaneously build content distribution hierarchies, without prior knowledge of the existence of other proxies. If a proxy lies along the natural path from an edge proxy to the origin server, it intercepts communication between them. Communication includes Web requests, as well as subscriptions for invalidation and replication. Such incremental actions by intermediate proxies builds sophisticated multi-level hierarchies rooted at origin servers. The interception is at the TCP layer. While the possibility of using Layer-4 transparent proxying for building hierarchies has been considered in [12], the scope of such an architecture is limited because of the problem that all packets of a TCP connection may not always follow the same path. If a transparent proxy intercepting a connection is unable to see all the packets of the connection, it cannot sanely proxy the TCP connection, which is a well known limitation [5]. A partial solution is to deploy transparent proxies at *focal points* within the network, which are guaranteed to see all packets of a connection. This makes the ad-hoc placement of proxies infeasible. SPREAD solves this problem by using what we call *Translucent* proxying, which guarantees that a proxy that sees the SYN of a TCP connection, will see all subsequent packets as well. This is accomplished by a novel use of IP tunneling and TCP-OPTIONS which we will discuss later.

2 SPREAD Architecture

The SPREAD architecture is based on a scalable content distribution network that spontaneously builds proxy caching hierarchies. In SPREAD, edge proxies connect to servers using a chain of proxies which are on the natural path from the edge proxy to the origin server (see Figure 1). Any given edge proxy may be a part of multiple proxy caching hierarchies rooted at different origin servers. In this section we discuss the basic principles that enable this. Unless otherwise mentioned, we assume a forward proxying scenario.

It is important to note that SPREAD is not concerned with how clients reach edge proxies. This is considered orthogonal to SPREAD. While this is indeed a non-issue in the case of forward proxies (which typically have a fixed or long-term mapping of clients to edge proxies), the reverse proxy scenario is trickier. With the advent of dynamic DNS tricks, the mapping of clients to edge proxies can be more fluid.

2.1 Spontaneous Hierarchies

A proxy caching hierarchy acts as an *application-level multicast* distribution tree [20], reducing the bandwidth usage in the network, the load at origin servers, and also reducing client latency. In the absence of a proxy caching hierarchy, origin servers need to directly communicate with all edge proxies, creating a huge burden on the origin server and the network. Using reliable multicast between the origin server and edge proxies would require an infeasible large number of multicast groups, and in addition reliable multicast is not yet available everywhere and has unresolved congestion control problems.

Proxy caching hierarchies already exist in the current Internet [2]. However, current hierarchies are static and

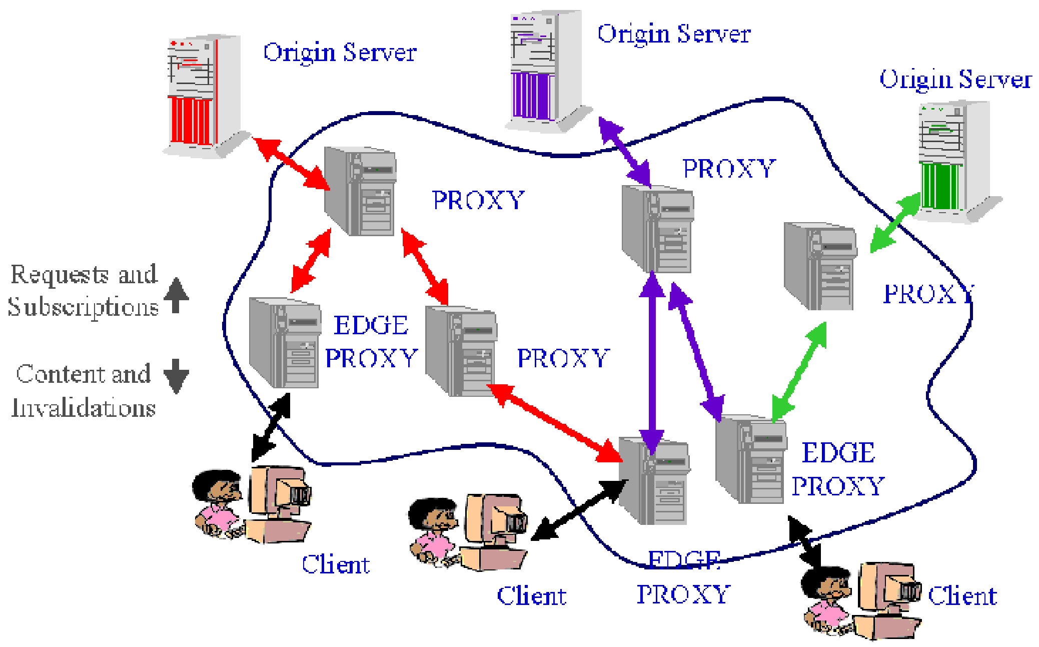


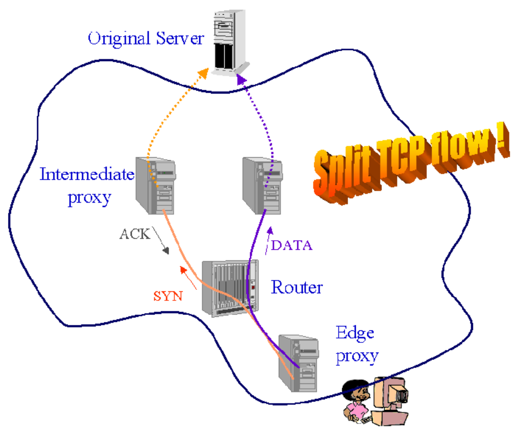
Figure 1: SPREAD Architecture

require substantial manual configuration and maintenance. To generate caching hierarchies that automatically configure themselves and forward packets to the origin servers through the shortest path routes, a routing architecture at the application level can also be implemented. Caches would then exchange application-level costs and calculate the best path to every origin server [15]. However, building an application level routing infrastructure is non-trivial, since route changes in the underlying network layer, will impact application-level routing. In contrast, SPREAD uses network layer routing and transparent proxies to build its proxy caching hierarchies. Requests travel from the clients to the origin servers following the shortest network path, and intermediate transparent proxies automatically pick up the connections for Web traffic (port 80). A transparent proxy that picks up a connection directly satisfies the document request if the document is stored in its cache, or lets the request travel towards the origin server if the document is not stored in its cache. As the request travels towards the origin server, the document request may be intercepted again by other transparent proxies, automatically forming a caching hierarchy. Changes in routes will create new hierarchies spontaneously, which will obey network level routing. No extra signaling is required to maintain the hierarchy.

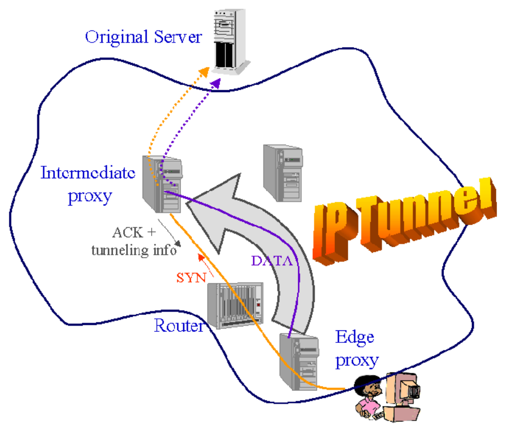
Naively building a hierarchy using transparent proxies is elegant, but has a serious problem. Since routing in an IP network can lead to situations where multiple paths from client to server may have the lowest cost, it can happen that packets of a connection follow multiple paths. In such a situation, a transparent proxy may see only a fraction of packets of the connection. Occasionally it is also possible that routes change mid-way through a TCP connection, due to routing updates in the underlying IP network. This problem limits the scope, requiring transparent proxies to be deployed exclusively at the edges or *focal* points within the network where they are guaranteed to see all the packets of the connection. SPREAD addresses this limitation by using Translucent Proxying, which allows the placement of Proxies *anywhere* in the network.

2.2 Translucent Proxying

Translucent Proxying Of TCP (TPOT) is a more sophisticated transparent proxying mechanism that allows proxies to be cascaded and networked together transparently, eliminating split TCP flows. Figure 2(a) provides a high level overview of the problem of split TCP flows and how Translucent Proxying solves the problem. When an edge proxy intends to connect with an origin server as shown in Figure 2(b), it issues a SYN packet, which reaches the intermediate proxy on the left. If the next packet of the TCP connection should be routed towards the proxy on the right, we have a situation where the proxy on the left cannot properly proxy the TCP connection. In Translucent Proxying, the proxy on the left sends back in the ACK, a signal to the edge proxy providing its IP address. The edge proxy will then use the IP address, to *tunnel* all remaining packets via the proxy on the left.



(a) Without Translucent Proxying.



(b) With Translucent Proxying.

Figure 2: Translucent Proxying solves the Split Flow problem using IP Tunneling.

Before describing the TPOT protocol, we provide a brief background of TCP/IP, which will help in better understanding TPOT.

Each IP packet typically contains an IP header, and a TCP segment. The IP header contains the packet's source and destination IP address. The TCP segment itself contains a TCP header. The TCP header contains the source port and the destination port that the packet is intended for. This 4-tuple of the IP addresses and port numbers of the source and destination uniquely identify the TCP connection that the packet belongs to. In addition, the TCP header contains a flag that indicates whether it is a SYN packet, and also an ACK flag and sequence number that acknowledges the receipt of data from its peer. Finally, a TCP header might also contain TCP-OPTIONs that can be used for custom signaling.

In addition to the above basic format of an IP packet, an IP packet can also be encapsulated in another IP packet. At the source, this involves prefixing an IP header with the IP address of an intermediate tunnel point on an IP packet. On reaching the intermediate tunnel point, the IP header of the intermediary is stripped off. The (remaining) IP packet is then processed as usual.

We now describe the TPOT protocol. Consider a source S that intends to connect with destination D via TCP, as shown in Figure 3. Assume that the first (SYN) packet sent out by S to D reaches the intermediary TPOT proxy T. (S,S_p,D,D_p) is the notation that we use to describe a packet that is headed from S to D, and has S_p and D_p as the source and destination ports respectively.

To co-exist peacefully with other end-points that do not wish to talk TPOT, we use a special TCP-OPTION "TPOT," that a source uses to explicitly indicate to TPOT proxies within the network, such as T, that they are interested in using the TPOT mechanism. If T does not see this option, it will take no action, and simply forwards the packet on

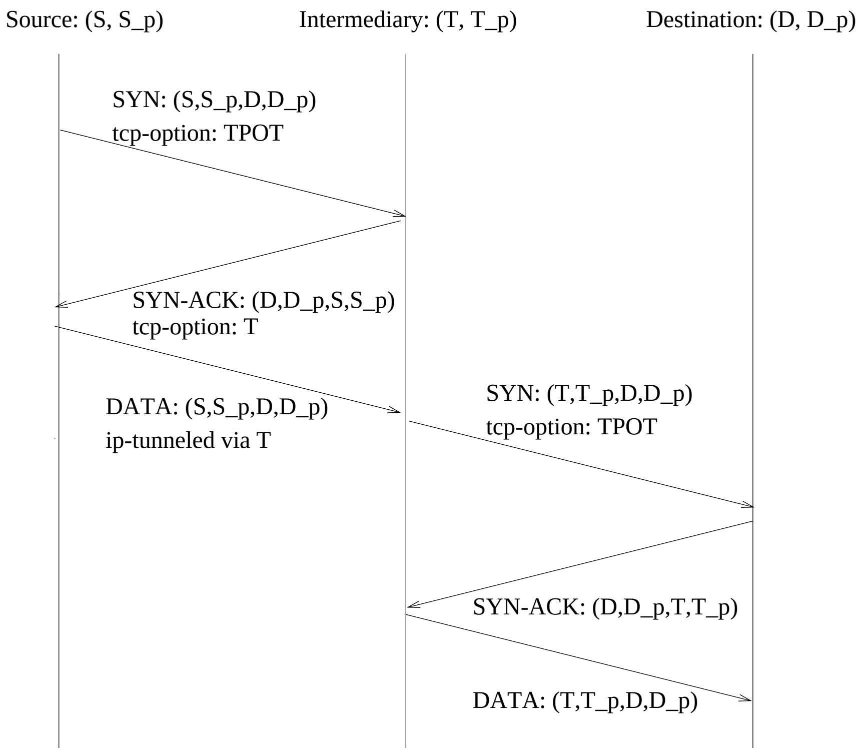


Figure 3: The Translucent Proxying Protocol

to D on its fast-path. If T sees a SYN packet that has the TCP-OPTION “TPOT” set, it responds to S with a SYN-ACK that encodes its own IP address T in the TCP-OPTION field. On receiving this packet, S must then send the remaining packets of that TCP connection, IP tunneled to T. From an implementation standpoint this would imply adding another 20 byte IP header with T's IP address as destination address to all packets that S sends out for that TCP connection. Since this additional header is removed on the next TPOT proxy, the total overhead is limited to 20 bytes regardless of the number of TPOT proxies intercepting the connection from the source to the final destination. This overhead can be further reduced by IP header compression [14] [10].

In SPREAD we use TPOT both for regular HTTP Requests as well as for subscriptions and unsubscriptions. Consider the case of a regular HTTP Request. For a cache hit, T is able to satisfy a request from S, and the response is simply served from one or more caches attached to T. In the case of a cache miss, T communicates with the destination D as shown in Figure 3. Note that the proxy T sets the TCP-OPTION “TPOT” in its SYN to D to allow possibly another TPOT proxy along the way to again proxy the connection. In Figure 3 we do not show such a scenario.

A more comprehensive description of the TPOT protocol, its variants, scalability and performance issues, as well as a prototype implementation may be found in [21].

3 Automated Content Distribution

In this section we describe basic content distribution in SPREAD. Edge proxies request objects from origin servers and requests are transparently intercepted by intermediate translucent proxy caches en-route to the origin server.

Proxies periodically calculate the expected number of requests per update period for every object, or for a volume (set of objects). Depending on the number of requests per update period, proxies may subscribe to invalidation or replication (see the Performance Optimization section). As the subscription travels to the origin server, an intermediate translucent proxy en-route intercepts the subscription (unless the intermediate proxy is overloaded -

in which case it lets the subscription pass through). On intercepting a subscription for invalidation, the intermediate proxy will subscribe itself to such a service, which in turn may be re-proxied by yet another proxy. Note that it is possible to limit this recursion by adding a hop-count field to the subscription, which gets decremented at each proxy. Once the counter hits zero, no other proxy will intercept it.

In the case where an invalidation (I) subscription arrives at a proxy, the proxy is forced to subscribe itself, unless of course it is already subscribed to I, or to Replication (R) - since R *implies* I. In the case where an R arrives at a proxy, it must subscribe to R, if it is not already subscribed to R. As we will see later, one may order mechanisms, in the increasing order V, I, R. If a child proxy finds a certain mechanism optimal, then a parent must, *at least*, use that mechanism. This assumes that children proxies are self-regulating as per SPREAD's optimal control policy. This will be discussed in a later section. Thus when a child proxy subscribes to I or R, all proxies on the path to the origin server are also automatically subscribed to *at least* that mechanism. Invalidations and replications themselves travel in the opposite direction. When an object is updated at the origin server, the server sends invalidations and/or replicas to proxy caches that are subscribed to I or R. Proxy caches that receive invalidations or replicas will themselves propagate the invalidations or replicas to children subscribed to I or R at the next tier. The process is repeated until the invalidation or document replica arrives at the edge proxy. Thus strong consistency is maintained.

3.1 Leases

Subscriptions have leases associated with them. On expire, a subscription must be renewed. These leases are set large enough so that repeated subscriptions do not overburden the network. At the same time, they are not so large that proxies commit themselves so far into the future when the changing statistics of the request and update rates suggest another mechanism. This is an implementation issue, that we do not discuss further.

3.2 State Information

Parent Proxies need to keep state information about the Children Proxies that are subscribed to invalidation or replication. However, the amount of state information required to keep track of subscribed Children Proxies is negligible compared to the disk capacity needed to store objects. Objects are usually subscribed and unsubscribed infrequently, and therefore, the amount of processing required is very small [24]. In addition, if multicasting is used, the load and state information at parent proxy caches is very small since only one object copy needs to be distributed to a set of Children Proxies. To further reduce the load and the state information, objects can be grouped into volumes at the cost of a coarser granularity for optimization and control. Here, a whole volume is invalidated or replicated instead of an individual object.

3.3 Reliability and Load Balancing

To ensure strong consistency even in the case of proxy cache failure, Parent Proxies periodically send *heart beats* to their Children Proxies. When a Parent Proxy dies, Children Proxies set the corresponding objects that the parent was responsible for as stale and re-send subscriptions towards to the origin server. The next (alive) proxies in the path to the origin servers then pick up the new subscriptions and become the new parents. This mechanism makes SPREAD reliable against even under catastrophic outages. A failed proxy or link, gracefully degrades the performance of SPREAD, without corrupting its correctness and guarantee of strong consistency.

Alternately, when a new proxy surfaces, it joins SPREAD incrementally. While existing subscriptions are not disturbed (since they are tunneled using TPOT to the existing parent), new subscriptions and Web requests that it sees can be proxied. Existing subscriptions also ultimately get re-proxied once their lease expires.

SPREAD automatically redistributes the load among its proxy caches, since every proxy cache is only responsible for those objects for which it sees requests, and then again only to its children. A last resort for an overloaded proxy server, is simply to stop intercepting any new Web requests and subscriptions, effectively going into *invisible* mode for all future services.

4 Optimizing SPREAD

To develop an appreciation for why and how SPREAD may optimize its performance, consider the scenario shown in Figure 4. An object is considered hotter than another if it is requested (read) more times than its is updated or modified (written).

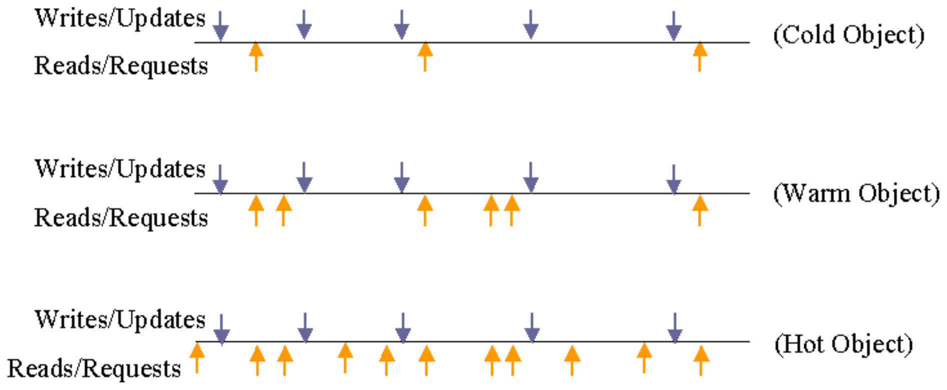


Figure 4: Cold, Warm and Hot objects.

Imagine that we want to minimize bandwidth consumption. For objects that are so cold, that every request appears after one or more writes/updates of the object, invalidations are useless, since every new object request sees a new object update. Replication, on the other hand, wastes even more bandwidth since objects are replicated on every write though they are rarely requested. In such a situation, it appears that client validation is probably the best policy. Note that what is important is the relative frequency of reads to writes. Objects that are hot, are objects for which there are one or more reads per write. In such a situation, replication is always preferred to client validation. Validation suffers from the problem that the second and future reads in an update/write interval will each require an *If-Modified-Since* poll, even if the object has not changed. The poll consumes bandwidth and causes additional delays. While invalidation performs better than validation, invalidation also wastes some bandwidth due to invalidation messages that perform no constructive function when compared to replication. Indeed, as we will see more rigorously later, invalidation is optimal for warm objects whose frequency of reads/requests is on the same order as the number of writes/updates. Note that in situations where not all three mechanisms are supported by the origin server, SPREAD will simply choose the best from what is available.

4.1 Analytical Model

We now build a mathematical framework to investigate how one might formulate the problem of deciding which mechanism to use for a given object. Since these choices will be made at each proxy, the issue of how a proxy estimates the various parameters relating to an object is an important one. These estimation issues will be dealt with in later sections.

We start with the case of an Edge Proxy that sees requests for some arbitrary object. We will extend our analysis to the case of an Intermediate Proxy (not just an Edge Proxy) in later sections. We assume that requests for the object from all the clients connected to an edge proxy cache are Poisson distributed with average request rate λ. The assumption of Poisson arrivals is a reasonable one [7], [3]. We also assume that objects are updated either periodically in a deterministic fashion, or randomly in an exponential distribution. This assumption will be discussed in a later section (Section 4.2). The average update period is denoted Δ.

We denote N to be the number of requests for the object per update period from all clients connected to the Edge Proxy. In the case when the object is updated periodically, the probability that there is at least one object request per update period from an Edge Proxy is then given by:

$$Pr\{N > 0\} = 1 - e^{-\lambda \cdot \Delta}$$

Note that $\lambda\Delta$ is the average number of requests per update period Δ.

When the object is updated exponentially, the probability that there is at least one object request per update period from an Edge Proxy is given by:

$$Pr\{N > 0\} = \frac{\lambda \cdot \Delta}{\lambda \cdot \Delta + 1}$$

To determine whether to use validation, invalidation, or replication, caches need to estimate the average number of requests per update period on a per-object basis. To calculate the average number of requests per update period, caches need to estimate i) the average request rate of an object and ii) the average update period of an object.

To estimate the request rate of an object, Edge Proxies can use the access logs from client access. The problem of estimating the request rate for an intermediate proxy is more involved (since it may not see direct hits from clients), and is discussed in Section 4.5. It is of course possible for Edge Proxies to inform intermediate proxies about their request rates (and in fact this was our initial design), but as we shall see later one can do without such communication.

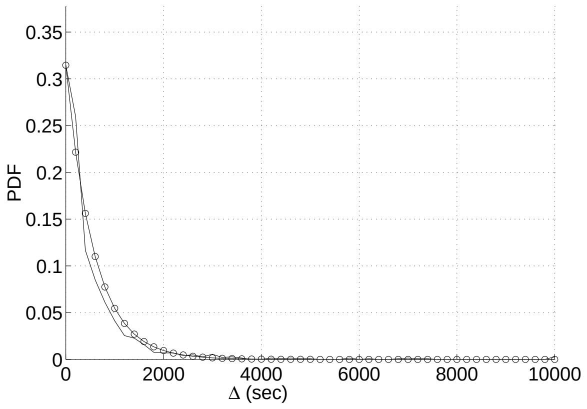
4.2 Estimating Update Rate

Proxy caches that are subscribed to invalidation (I) or replication (R) for an object, see all updates, and can therefore estimate the update rate in a straight-forward fashion. However estimating the update period of an object that uses validation (V) is more complex. Since the proxy can only inspect the *Last-Modified* time of an object when it is requested, information on updates that are never requested are lost. However, proxies can use the difference between the time of a request (or Date field) and the Last-Modified time, to infer the average update period of an object if they know the probability distribution of object updates.

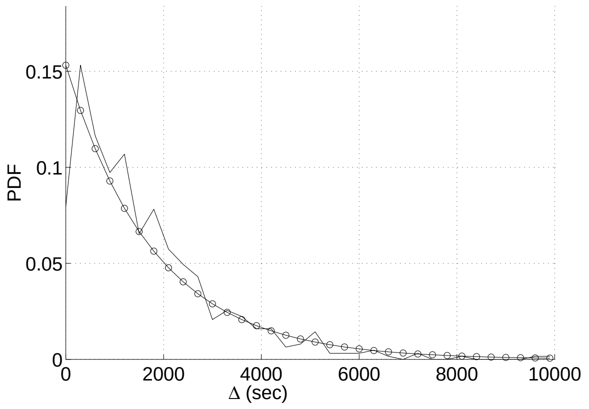
We should point out that headers such as the ”Expires” header which explicitly provide consistency information, cannot be used here for two reasons. First, our own study of the Web and those of others have shown that most cacheable documents have their Expires headers at a value that effectively makes the TTL zero anyway. Further, such protocol headers (even when non-zero) do not provide realistic values for update rates, since, these headers only need to provide a lower bound. In other words, a document whose TTL is set to 10 seconds (via an Expires header or some other metadata) may update itself after 10 days, and yet be perfectly in line with the HTTP protocol.

Previous work on the distribution of object updates suggested that objects are approximately updated randomly following an exponential distribution or periodically [6]. However, these results were performed with client traces that did not see all server updates. To better study the distribution of object updates we polled different sites once every minute for a period of 10 days, recording the last-modified-time stamp of the object on every poll. Then we calculated the update period of an object as the time difference between two different last-modified-time stamps. This experiment gave us the real update pattern of an object within a resolution of one minute. Our results confirm the ones presented in [6]. We found that there are a large number of Web sites that update their information periodically, e.g. every 15 or 30 minutes. However, we also found a large number of Web sites that update their Web sites randomly following an exponential distribution. In Figure 5 we present the distribution of object updates for two different news sites. We clearly see that the distribution of object updates in both sites approximates an exponential distribution.

Note that Proxy Caches can easily determine if an object is updated periodically or is exponentially distributed by studying the variance of object updates. Once they have determined if the object is updated periodically or exponentially, they can use the time difference between object requests and the last-modified-time stamps to estimate the average update period [11].

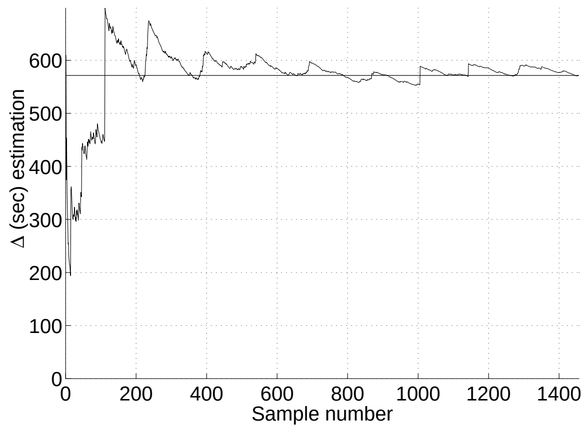


(a) Spanish Newspaper Web Site

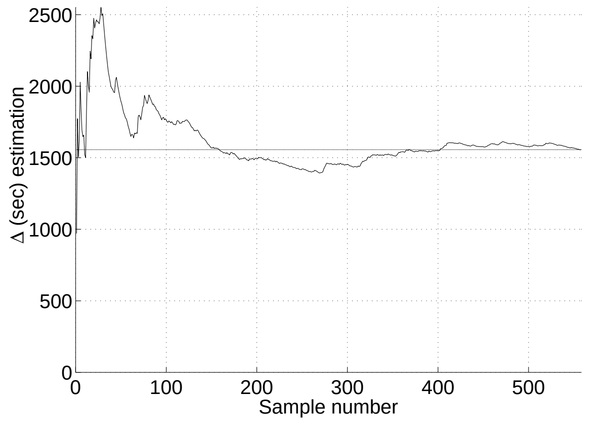


(b) BBC News Web Site

Figure 5: Distribution of object update intervals. 10 day logs. Servers are polled every minute.



(a) Spanish Newspaper Web Site



(b) BBC News Web Site

Figure 6: Estimation of the average update interval as a function of the number of samples

Figure 6 shows how rapidly the estimate of the average update period converges with the number of samples. Each sample measures the time difference between every request and the last-modified-time, and computes a simple average. We observe that after 200 samples, the estimate of the average update period converges to 600 seconds in Figure 6(a), and 1500 seconds in Figure 6(b). This warm-up time is small enough to make such estimators viable.

4.3 Optimizing Bandwidth

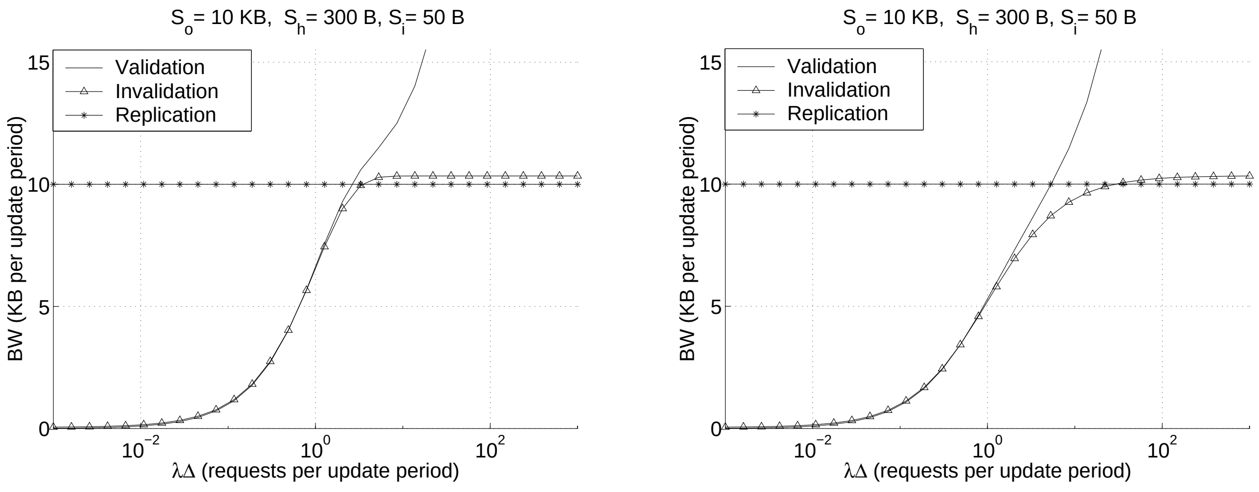
Next, we compute the bandwidth usage by each mechanism to deliver up-to-date content. We define the bandwidth usage B as the average number of bytes consumed per update period Δ in a proxy's link.

Let S_o be the actual size of a Web object. Let S_h be the size of an HTTP header, which is considered to be the same as the size of an IMS request. Let S_i the size of an invalidation message.

The bandwidth usage per Web object for validation B_V, invalidation B_I, and replication B_R can be easily shown to be:

$$
\begin{aligned}
B_V &= Pr\{N > 0\} \cdot S_o + E[N] \cdot S_h \\
B_I &= Pr\{N > 0\} \cdot (S_o + S_h) + S_i \\
B_R &= S_o
\end{aligned}
$$

Note that in our analysis we have assumed that the object has no max-age or expires header set by the server. In a situation where the server would set a max-age or an expires header different than zero, the analysis would need to be modified accordingly, though the qualitative results of our paper would still hold.



(a) Periodic Updates (b) Exponentially Distributed Updates

Figure 7: Bandwidth usage

Figure 7 shows the bandwidth usage of validation, invalidation, and replication depending on the average number of requests per update period $\lambda \cdot \Delta$. The values for S_i, S_h and S_o are representative of what is typical for the Web today. For objects with few requests per update period, replication wastes a lot of bandwidth compared to validation or invalidation, since the object is preloaded into the caches even when it is not requested by the clients. On the other hand, validation and invalidation have a low bandwidth usage since the object is only fetched into the caches when it is requested by a client. For a large range of values for $\lambda \cdot \Delta$ from about 0 to 1 requests per update period, validation uses slightly less bandwidth than invalidation since every request finds a new object update and therefore the overhead of IMS requests to the origin server is almost zero. For values above about 1 requests per update period, replication does well, trailed by invalidation, which suffers because of the extra invalidations that are sent out. Validation works poorly, due to the fact that every request generates an IMS request which is typically much heavier than an invalidation. Figure 7 shows that the mechanism that consumes the least bandwidth is different in different regimes of $\lambda \cdot \Delta$, and that the order in which the different policies are optimal is V, I, and R as $\lambda \cdot \Delta$ increases.

4.4 Switching Thresholds

Let the switching thresholds between V and I, and I and R, be denoted by Th_{VI} and Th_{IR} respectively. Table 1 shows the thresholds to switch among the different policies at an edge proxy.

From Table 1 it is easy to prove that for all reasonable values of S_i, S_h and S_o, we have the property that: $Th_{VI} < Th_{IR}$. Further, since λ increases as one moves closer to the origin server, we have the property that at any level of the hierarchy if a given mechanism is optimal for a proxy, it must be *at least* good for the parents above. That is:

Perspective	Th_{VI} (req per update period)	Th_{IR} (req per update period)
Edge Proxy (Deterministic Update Period)	$\sqrt{\frac{2 \cdot S_i}{S_h}}$	$ln(\frac{S_o+S_h}{S_h})$
Edge Proxy (Exponential Update Period)	$\frac{S_i+\sqrt{S_i^2+4S_hS_i}}{2 \cdot S_h}$	$\frac{S_o-S_i}{S_h+S_i}$

Table 1: Thresholds to switch between validation and invalidation Th_{VI}, and between invalidation and replication Th_{IR}

- if a proxy finds V optimal, then its parent may find V, I, or R optimal.
- if a proxy finds I optimal, then its parent may find I, or R optimal.
- if a proxy finds R optimal, then its parent will find only R optimal.

By the above result, if a proxy subscribes to a certain policy, it must also be in its parent's best interest to *at least* have that policy in place. Therefore it always makes sense to proxy subscriptions on behalf of a child proxy. This clearly validates SPREAD's design model, even if by serendipity.

4.5 Estimating Request Rate at an Intermediate Proxy

As we discussed earlier, estimating the request rate of an object at an Intermediate Proxy may be complicated because it does not see direct hits from clients. However, we argue here that given the observations of the previous section, this can be substantially simplified by breaking down the possibilities into two cases.

- Case 1: If *any* of the children are in the R state, then, the parent proxy is also in the R state and cannot go to I until all of its children unsubscribe from R. No decision need be made by the proxy, and therefore estimating request rate is not essential. (Note that when the last child proxy unsubscribes from R, we can seed the estimator with the estimated request rate from that child to be $\frac{Th_{IR}}{\Delta}$).
- Case 2: In this case, children proxies are in the I or V state. For those in the V state the estimation of the request rate is straightforward, since the proxy sees all the requests (HTTP GETs or IMS requests). For proxies in the I state, the request rate may be computed in a more sophisticated fashion. Here, the proxy estimates the time interval between an invalidation and the immediate following request. For both exponentially distributed and periodic (deterministic) update periods, we may compute an estimate for the request rate from that child proxy using standards results for residual life from the area of Renewal Theory [4]. For reasons of space we omit a lengthier discussion.

4.6 Latency

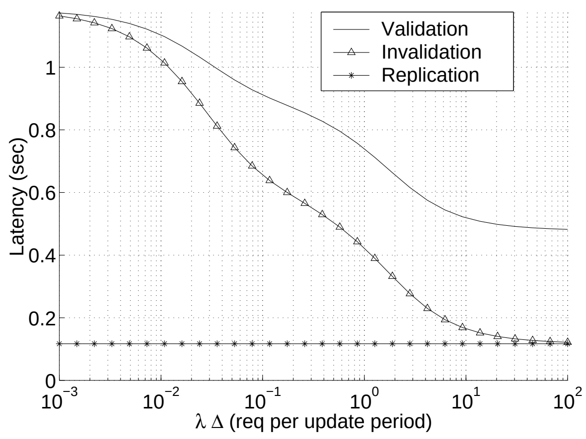
In this section we investigate the latency experienced by the clients when validation, invalidation, or replication are used. Let t_{os} be the transmission time of an object when it is retrieved from the origin server. Let t_{pc} and t_{cc} be the transmission time of an object when it is transmitted from the Parent Proxy and from the Children Proxies respectively. Let RTT_{os} be the round-trip-time between the origin server and any proxy cache. The expected latency experienced by a client depends on the tree level where the object is hit. Let L be the number of links traversed to find a object. In this section, we consider a simple two-tier caching hierarchy, however, the analysis can be easily extended for a different number of cache tiers. The exact calculation of the probability distribution function of L can be found in [20] and has been omitted due to space limitations. Given the distribution of L we can calculate the expected latency experienced by a client for validation T_V, invalidation T_I, and replication T_R as:

$$T_V = Pr\{l = cc\} \cdot (t_{cc} + RTT_{os}) + Pr\{l = pc\} \cdot (t_{pc} + RTT_{os}) + Pr\{l = os\} \cdot t_{os}$$

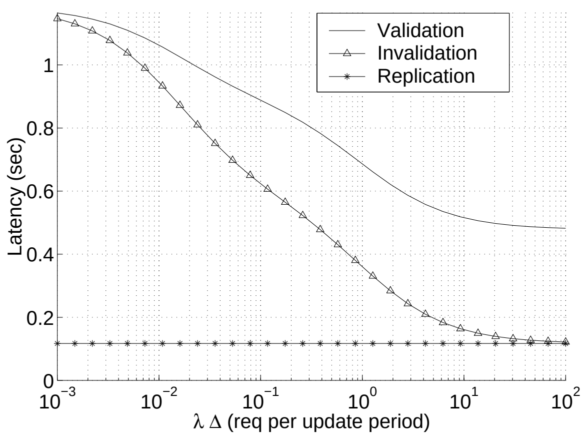
$$
\begin{aligned}
T_I &= Pr\{l = cc\} \cdot t_{cc} + Pr\{l = pc\} \cdot t_{pc} + Pr\{l = os\} \cdot t_{os} \\
T_R &= t_{cp}
\end{aligned}
$$

To consider real values for the latency, we analyzed 10 days of logs on the local proxy at Eurecom, which is connected to a caching hierarchy through a Parent Proxy. We averaged the latencies during the 10 days of the trace to obtain the following values:

- Transmission time from the local proxy: T_{cc} = 117 msec,
- Transmission time from a Parent Proxy: T_{pc} = 585 msec,
- Transmission time from the origin server: T_{os} = 1183 msec,
- Round-Trip-Time to the origin server: RTT_{os} = 300 msec.



(a) Periodic Updates



(b) Exponentially Distributed Updates

Figure 8: Expected Latency

We considered the case of a caching hierarchy with 64 children caches and a single parent cache. Based on these values Figure 8 shows the latency experienced by a client for validation, invalidation, and replication. Using replication, clients always experience small latencies since the Edge Proxy always has the object replicated to it. This, as we have seen earlier, may be extremely wasteful of bandwidth. As the number of requests per update period increases, the probability of finding an object at proxies closer to the client increases, thus, reducing the latency experienced. Invalidation offers better latencies than validation since client requests do not need to contact the origin server every time. However, for invalidation to provide similar latencies as replication, the number of requests per update period needs to be very high (i.e. approx. 100 requests/update period). For such popular objects, using invalidation to reduce client's latency is not the best option since replication generates slightly less traffic in the network (see Figure 7), providing very small latencies for *all* receivers.

4.7 Multicast Extensions

In this section we consider the case when the network supports multicasting. If multicasting is available, Parent Proxies may decide to multicast invalidations and replicas to their Children Proxies instead of sending them via unicast. For validation, objects and IMS messages are distributed via unicast. For invalidation, the actual object is fetched via unicast by the Children Proxies, however, invalidation messages are multicast to all proxy caches. For replication, object updates are pushed via multicast from the parent proxy cache to all Children Proxies.

The decision to use multicast or unicast depends on the multicast gain $G = \frac{C_{mc}}{C_{uc}}$, that is the multicast cost C_{mc} divided by the unicast cost C_{uc}, which is a function of the network topology, the number of Children Proxies and their location. Several studies have shown that the multicast gain in a wide range of network topologies can be approximated by $G{=}M^{-0.2}$, where M is the number of receiving proxies [18]. Therefore, it is enough for a Parent Proxy to know the number of subscribed Children Proxies to estimate the multicast gain and therefore decide whether to turn on multicast or not.

The bandwidth usage in the network of validation B_V, invalidation B_I, and replication B_R to deliver one byte from a Parent Proxy to the Children Proxies with multicast is:

$$\begin{aligned}
B_V &= Pr\{N > 0\} \cdot S_o \cdot C_{uc} + E[N] \cdot S_h \cdot C_{uc} \\
B_I &= Pr\{N > 0\} \cdot (S_o + S_h) \cdot C_{uc} + S_i \cdot C_{mc} \\
B_R &= S_o \cdot C_{mc}
\end{aligned}$$

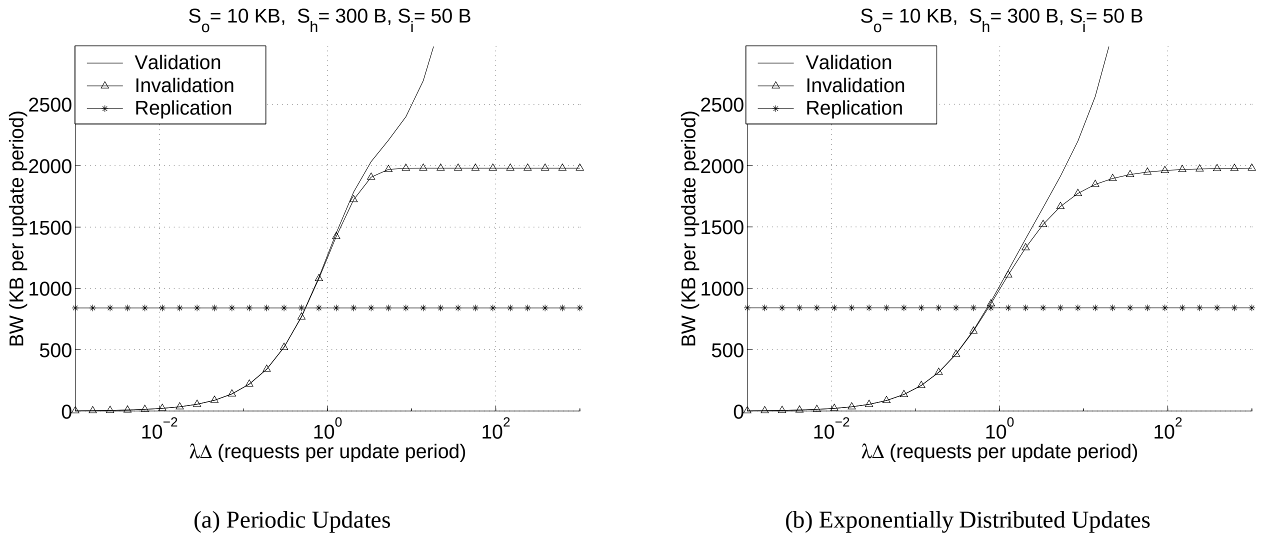


Figure 9: Bandwidth usage with multicast enabled

To study the effect of a multicast distribution we analyze the case where the network connecting the Parent Proxy with its M children proxy caches is a full O-ary tree with height H [19] (a full O-ary tree has proved to be a good model for network topologies, providing very realistic results [16]). In Figure 9 we present the bandwidth usage inside the network for validation, invalidation, and replication when multicast is enabled.

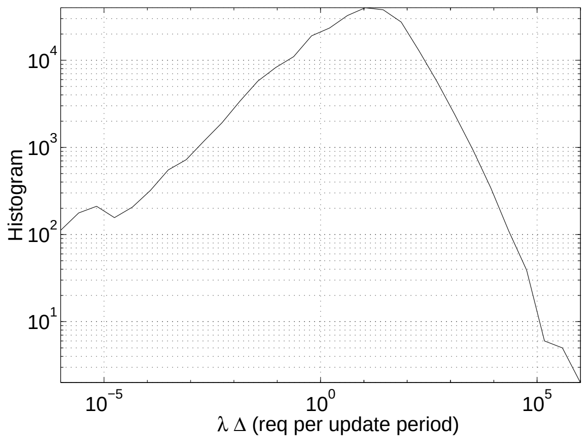
Comparing Figures 7 and 9 we observe that the relative performance of validation is not modified since validation does not benefit from the fact that multicast is enabled. We also observe that the relative performance of invalidation is slightly smaller since invalidation messages are now multicasted. For replication, the bandwidth savings are very high, since the cost of replication is small. Of course, the multicast gain depends on the network topology and the number of receivers; however, even in the worst case a multicast distribution performs no worse than unicast, and the relative performance of validation, invalidation, and replication would then be the same as the one in Figure 7.

5 Trace-driven Simulation

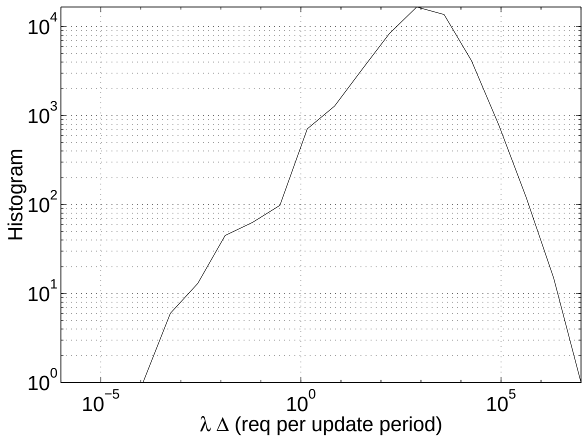
Based on the switching thresholds calculated in Section 4.4, we now perform a trace driven simulation to get a feel for how SPREAD will behave in a real-life setting. To that end, we analyze log traces from one access node

(POP) at AT&T Worldnet (Bridgeton) over a period of 10 days, collected in May 1999. The total number of requests in the trace is roughly 10 million. 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 to study how the control algorithm we use in SPREAD will perform. These two object types constitute an overwhelming majority (over 90%) of the accesses. 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, which is the average update period in the case of exponentially distributed update periods, and is equal to the half of the average update period for periodic updates. In reality, a SPREAD proxy would continuously monitor the request and update rates; however, using the average update period during the trace was a suitable approximation for the purpose of our simulation study.

In Figure 10(a) we show the distribution of objects of type text/html that have a certain number of requests per update period. We see that most objects have a value which is concentrated between 10^{-4} requests per update period and 10^{4} requests per update period.



(a) Distribution of text/html.



(b) Distribution of image/gif.

Figure 10: Distribution of requests per update period $\lambda \cdot \Delta$.

Combining the results presented in Figure 10(a) and Table 1, we can calculate the percentage of objects that would use validation, invalidation, or replication to minimize the bandwidth usage.

Table 2 shows the percentage of objects requiring every scheme in the case of periodic updates, and the value of the switching points in terms of requests per update period (Th_{VI} and Th_{IR}) for a sample HTML document of size 10KB.

Perspective	Threshold (req per update period)	V	I	R
Bandwidth (Periodic)	Th_{VI}=0.7, Th_{IR}=3.6	18.4%	19%	62.2%
Bandwidth (Exponential Distribution)	Th_{VI}=0.55, Th_{IR}=29	16%	52%	32%

Table 2: Percentage of objects that require validation (V), invalidation (I), and replication (R). Periodic and Exponentially distributed Updates

From Table 2, we see that in the case of periodic updates, 19% of the objects would require invalidation to minimize bandwidth usage, and 63% would require replication. In the case of exponentially distributed update periods we also calculated the percentage of objects that would require each scheme, and see that the percentage of objects that would require invalidation increases to 86%.

In the Optimization section, we calculated the bandwidth usage in a proxy's link for a single document with varying requests per update period $\lambda \cdot \Delta$, and the average client's latency for a simple two-tier cache hierarchy. Next, we calculate the total average bandwidth usage and the expected latency for validation, invalidation, replication, and SPREAD. We sum up the bandwidth used by *all* objects, and scale the bandwidth we obtain per update period - to per second - by dividing the result by the object's update period Δ.

Table 3 summarizes the results for bandwidth usage, and the corresponding client latency.

Perspective	V	I	R	SPREAD
Bandwidth (KB/sec)	1.6	5.3	803	1.4
Client Latency (sec)	0.58	0.28	0.11	0.26

Table 3: Bandwidth consumption and resulting Latency for validation (V), invalidation (I), replication (R), and SPREAD for HTML documents. Periodic Updates.

From Table 3 we see that the bandwidth needed to deliver all documents with validation is quite small since most of the documents in the trace have few requests per update period. Invalidation, on the other hand, has a higher bandwidth usage than validation, since invalidation messages are sent for documents that are never requested. Replication has the highest bandwidth usage since all documents are being replicated, and many are not requested. SPREAD, has the minimum bandwidth usage since proxies automatically select validation, invalidation, or replication to optimize bandwidth. The benefits in terms of bandwidth of SPREAD compared to validation are not very high since there are not many hot documents in the trace that produce a large number of IMS requests (the bandwidth usage of validation would be much higher in the case of more popular documents). However, the latency experienced by the clients with SPREAD is about half the latency experienced with validation. Even though SPREAD is not optimized to minimize latency we see that the latency offered by SPREAD is smaller than for validation or invalidation. As SPREAD proxies subscribe to invalidation or replication to minimize bandwidth usage, the latency reduces, since the origin server is not contacted so often. Of course, replication has the lowest latency at the cost of high bandwidth usage. We have also calculated the same parameters than in Table 3 for the case of exponentially update periods, and the results for exponentially update periods do not differ much from those for periodic updates.

Next, we also study the case for objects of type image/gif (see Figure 10(b)). GIF objects tend to change less frequently, and therefore the number of requests that a GIF object receives before it is updated is much higher than for HTML objects (Figure 10(a)). Table 4 shows the total bandwidth usage by GIF objects using validation,

Perspective	V	I	R	SPREAD
Bandwidth (KB/sec)	7.6	2.4	72	1.6
Client Latency (sec)	0.45	0.14	0.11	0.12

Table 4: Bandwidth consumption and resulting Latency for validation (V), invalidation (I), replication (R), and SPREAD for GIF images. Periodic Updates.

invalidation, replication, and SPREAD. From Table 4 we see that validation performs worse than it does for HTML documents, since validation results in a higher number of IMS queries to the origin server (since GIFs see more requests per update). Replication performs better than it does for HTML documents for the same reason. This also causes SPREAD to improve on invalidation much more than it did with HTML. As before, we see that though SPREAD is tuned to optimize bandwidth, it has an average latency which is very close to that achieved with replication.

Finally, in Table 5 we add the total bandwidth usage and calculate the average latency for text/html and image/gif objects to see how the various schemes perform. We see that the bandwidth savings and the reduction in latency for SPREAD compared to validation, invalidation, and replication are much more relevant than for either text/html or

Perspective	V	I	R	SPREAD
Bandwidth (KB/sec)	9.2	7.7	875	3
Client Latency (sec)	0.49	0.18	0.11	0.16

Table 5: Bandwidth consumption and resulting Latency for validation for validation (V), invalidation (I), replication (R), and SPREAD for HTML documents and GIF images. Periodic Updates.

for image/gif objects alone. That is, while one of the mechanisms may be suited for one type of object, e.g. validation to reduce bandwidth usage for text/html or invalidation for image/gif, SPREAD does well overall, distancing itself from the other mechanisms when a mix of objects are considered.

6 Conclusions and Future Work

In this paper we introduced SPREAD, a new architecture for content distribution. SPREAD uses a network of proxies that automatically configure themselves and make autonomous decisions on how to maintain cache consistency. They dynamically choose between client validation, server invalidation and replication to optimize bandwidth usage. One key component of SPREAD is that it uses a new class of Transparent proxies called Translucent proxies. Translucent proxies can be cascaded and networked together transparently, without requiring them to be placed at focal points in the network.

SPREAD is also showing promise as a base platform for a large set of other wide-area applications for which self-organization, scalability and robustness are important. To explore this further, we are currently pursuing the use of SPREAD for reliable multicast and, for broadcasting content.systems.

7 Acknowledgments

We thank Anja Feldmann for collecting and providing traces from AT&T Worldnet.

8 Vitae

Pablo Rodriguez Pablo Rodriguez is a senior graduate student at the Institut EURECOM, finishing up his thesis on Scalable Content Distribution in the Internet. He has been active in the areas of Web caching and replication, satellite dissemination of Web documents, caching infrastructures for delivering up-to-date content, and scalable broadcasting solutions.

Sandeep Sibal Dr. Sandeep Sibal is a Senior Technical Staff Member in the Internet and Networking Systems Center at AT&T Labs – Research. His general interests are in Internet technologies and services, and he is currently working on topics in Content Distribution and Layer-4 proxies.

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] M. F. Arlitt and C. L. Williamson, "Web ServerWorkload Characterization: The Search for Invariants", In *Proceedings of the ACM SIGMETRICS*, New York, May23–26 1996.

[4] D. R. Cox, “Renewal Theory”, 1962.

[5] P. Danzig and K. L. Swartz, “Transparent, scaleable, fail-safe Web caching”, Technichal report, Network Appliance. Santa Clara, CA, USA, 1999.

[6] F. Douglis, A. Feldmann, B. Krishnamurthy, and J.Mogul, “Rate of change and other metrics: A live study of the World Wide Web”, In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[7] 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.

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

[9] J. Gwertzman and M. Seltzer, “World-Wide Web Cache Consistency”, In *Proc. 1996 USENIX Technical Conference*, San Diego, CA, January 1996.

[10] V. Jacobson, “Compressing TCP/IP Headers for Low-Speed Serial Links”, RFC 1144, 1990.

[11] L. Kleinrock, *Queuing Systems, Volume I: Theory*, Wiley, 1975.

[12] P. Krisnan, D. Raz, and Y. Shavitt, “Transparent En-Route Caching in WANs”, In *Work-in-progress in the 4th International Caching Workshop*, San Diego, March 1999.

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

[14] B. N. M. Degermark and S. Pink, “RFC 2507: IP header compression”, Feb 1999.

[15] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson, “Adaptive Web Caching: towards a new global caching architecture”, In *3rd International WWW Caching Workshop*, June 1998.

[16] J. Nonnenmacher and E. W. Biersack, “Performance Modelling of Reliable Multicast Transmission”, In *Proc. IEEE INFOCOM’97*, Kobe, Japan, April 1997.

[17] F. Overview, “Sandpiper, Thousand Oaks, CA, USA. Oct 1999”.

[18] G. Phillips, S. Shenker, and H. Tangmunarunkit, “Scaling of Multicast Trees: Comments on the Chuang-Sirbu Scaling Law”, In *ACM SIGCOMM’99*, volume 29, Harvard University, Massachusetts, USA, September 1999.

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

[20] 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.

[21] P. Rodriguez, S. Sibal, and O. Spatscheck, “TPOT: Translucent Proxying of TCP”, Technical report TR 00.4.1, AT&T Research Labs, 2000.

[22] D. Wessels, “Squid Internet Object Cache: http://www.nlanr.net/Squid/”, 1996.

[23] K. Worrel, “Invalidation in large scale network object caches”, Master’s Thesis, University of Colorado, Boulder, 1994.

[24] H. Yu, L. Breslau, and S. Shenker, “A Scalable Web Cache Consistency Architecture”, In *Proceedings of ACM SIGCOMM’99*, Cambridge, sep 1999.