Proactive Caching of DNS Records: Addressing a Performance Bottleneck

Edith Cohen
AT&T Labs–Research
Florham Park, NJ 07932 USA
edith@research.att.com
http://www.research.att.com/~edith

Haim Kaplan
Tel-Aviv University
Tel Aviv, Israel
haimk@math.tau.ac.il
http://www.math.tau.ac.il/~haimk

Abstract

The resolution of a host name to an IP-address is a necessary predecessor to connection establishment and HTTP exchanges. Nonetheless, DNS resolutions often involve multiple remote name-servers and prolong Web response times. To alleviate this problem name servers and Web browsers cache query results. Name-servers currently incorporate *passive* cache management where records are brought into the cache only as a result of clients' requests and are used for the TTL duration (a TTL value is provided with each record). We propose and evaluate different enhancements to passive caching that reduce the fraction of HTTP connection establishments that are delayed by slow DNS resolutions: (A) *Renewal policies* refresh selected expired cached entries by issuing unsolicited queries. Trace-based simulations using Web proxy logs demonstrated that a significant fraction of cache misses can be eliminated with a moderate increase in the number of DNS queries. (B) *Simultaneous-validation* (SV) transparently uses expired records. A DNS query is issued if the respective cached entry is no longer fresh, but concurrently, the expired entry is used to connect to the Web server and fetch the requested content. The content is served only if the expired records used turn out to be in agreement with the query response. [1]

I. Introduction

The resolution of a host name to an IP-address is a necessary predecessor to communication between Internet hosts. In particular, it is required for connection establishment and HTTP exchanges with a Web server. DNS (Domain Name System) [21], [22] is in essence a distributed database that answers queries on mapping between names and addresses. Name-servers belong to a hierarchy where typically servers responsible for large domains delegate other name-servers to be in charge of subdomains. DNS was designed prior to the onset of the Web, but fortunately, its design allowed it to scale and accommodate the explosive growth of the Internet. The perhaps unavoidable downside of this design is that resolving a DNS query often involves communication with at least one remote name-server, and may require following a delegation/referral chain of several remote name-servers. Furthermore, since name-servers are different hosts than the HTTP servers that are contacted subsequently, DNS resolutions create additional potential point(s) of failure. Resolutions typically use UDP exchanges and use timeout and retransmission, which adds a delay on the order of seconds, in the event of packet loss.[2] The overall impact of DNS resolutions on user-perceived latency stems from both additional RTTs to remote servers and sensitivity to long timeouts.

Caching of query results at local name-servers decreases both overhead and user-perceived latency and is instrumental for performance. A caching mechanism for DNS was specified in RFCs 1034 and 1035 [21], [22], and is integrated in BIND [1], the most popular name-server software. Name-servers resolve client queries about hostnames inside and outside their authoritative zones. Outside queries are resolved through communication with other name-servers, following referral chains to an authoritative name-server. Name-servers cache the results of queries sent to other name-servers. Each resource record (e.g., CNAME, IP-address, or an authoritative name-server) is provided with a TTL (Time To Live) value, and name-servers may cache entries they are not authoritative for only until the TTL expires.

Although "cache misses" (resolutions necessitating exchanges with remote servers) precede only a fraction of HTTP connections, their durations tend to be unpredictable and heavy-tailed. Studies show that when long Web waits do occur, DNS resolutions are a significant cause [7], [17], [23], [25]. Such occasional unexpected long delays significantly impact the consistency of service quality, which is often measured by extremes. High variance in "connecting" time

[1] Preliminary version of this paper appeared in SAINT'01 [5]

[2] In contrast, if packet loss occurs during a TCP connection, the timeout value is set adaptively according to previous history of RTT duration, which is typically considerably shorter than the default value.

was not critical for applications such as email, telnet, and FTP, that dominated the Internet when DNS specifications emerged, but is detrimental to Web browsing. Unfortunately, it seems that the impact of DNS resolutions is inherent in the current architecture. As bandwidth increases and content transmission time decreases, Web service speed would be increasingly dominated by RTTs, and the relative contribution of DNS resolutions would only increase.

Caching in BIND currently works in a passive manner: information for which a name-server is not authoritative for is obtained only as a consequence of a client query and is cached until either the TTL expires, or the name-server process dies. Here we propose and evaluate enhancements to basic passive caching aimed at reducing user-perceived latency due to DNS query time. We suggest making DNS caches proactive. *Proactive DNS caching* integrates automatically-generated "preemptive" queries that update the cache. These automatic refreshes make it more likely that a forthcoming user request would find a fresh cached answer and would not trigger communication with remote name servers. The challenge is to balance the number of *eliminated cache misses* and *overhead* of additional DNS queries issued to remote name-servers. *Renewal policies*, proposed and studied here, are a natural class of proactive caching algorithms addressing this challenge. Our enhancements fall within the framework of the current DNS architecture and can be locally deployed.

Renewal Policies

Renewal policies refresh cached entries upon their expiration by issuing a new query. The different policies vary by when an entry is renewed. We consider several natural policies, based on reference locality (analogous to the cache replacement algorithm LRU), access-frequency (analogous to LFU), and an adaptive per-hostname policy (analogous to policies studied in [6], [10], [18]). These analogies are made based on properties of the request sequence that are exploited but the underlying cost/benefit measures are different. We experimentally evaluated and compared performance of the different polices using a heterogeneous set of proxy logs, consistently obtaining significant increase in hit-rate at reasonable overhead costs. [3]

Our study shows that the best renewal policies eliminate about 60% of the DNS misses at the cost of roughly doubling the number of DNS queries. Section III indicates how this reduction in the miss rate of the DNS server translates to a reduction in user-perceived latency. The measurements presented in Section III show that DNS misses are responsible for about 25% of the HTTP requests which take over 300ms.[4] Thus by eliminating 60% of the DNS misses we are likely to reduce the fraction of HTTP requests which exceed 300ms from 40% to about 34%. Notice that downloading a single Web page triggers about 10–20 HTTP requests on average to all embedded contents [20], [23]. Thus a reduction of 5% in the fraction of long HTTP requests translates to a larger reduction in the fraction of Web pages that have long download time.

Since DNS traffic constitute a small fraction (3%) of the overall network traffic [24] we expect that the cost of increasing the number of DNS queries by a factor of two would not have a detrimental global effect. Highly-loaded DNS servers may suffer from an increase in the number of queries which they get. Since system administrators control the load on their name servers by varying TTLs and allocating secondary servers we suspect that load on majority of the DNS servers is not a bottleneck.[5] Those servers that are highly loaded, like the root servers for example, can be backed up by more machines to distribute the load.

Renewal vs. Preresolving

Preresolving (prefetching DNS queries) is a technique related to renewal. Preresolving was proposed in [7] as a low-overhead alternative to the prefetching of documents. Preresolving applies prediction schemes (e.g., by analyzing hyperlinks or tracking access patterns) to decide which hostnames (Web servers) to preresolve. The work [7] demonstrated a potential for considerable reduction in user-perceived latency when preresolving hostnames returned on search-engines responses. The tradeoff of latency and overhead also measures the performance of preresolving algo-

[3]We used the proxy logs as a source for a real request sequence of a relatively large population of Web users. In our simulation we replayed the request sequence on a different environment than where it was logged. In that sense we did not simulate a real scenario. We believe however that our simulation is indeed representative of DNS and HTTP performance in real Web proxies. Similarities between our findings and other recent results (which we describe below) back up this assumption. We simulated only DNS traffic generated by Web transactions as our main application was reducing Web latency.

[4]Excluding requests carried over persistent connections.

[5]TTLs are made very short to do load balancing only when there is an infrastructure to support the additional queries, like in CDNs for examples.

rithms. The basic difference between renewal policies and preresolving is in their deployment: Preresolving utilizes predictions made based on per-user access patterns and currently-viewed hyperlinks. This information is available at the user's browser or proxy server, and therefore, preresolving queries would most naturally be initiated there and be viewed at the local name-server as regular client queries. Renewal policies, on the other hand, aggregate per-record patterns from DNS query sequences. Hence, they can be incorporated within the name-server cache and be transparent to its clients. Preresolving and renewal also differ in their impact on traffic: first, like document prefetching, preresolving is more likely to generate bursts [12]. Second, prediction-based preresolves are more likely to include loaded root servers whereas renewals are often directed only to lightly loaded servers lower in the DNS hierarchy.

Simultaneous Validation

DNS TTL-based freshness control poses an inherent conflict for a domain administrator assigning TTL values. Smaller TTL values increase user-perceived latency and name-server load, and makes the site more likely to be inaccessible when the name server is down. Large TTL values, on the other hand, constitute long-term commitments. If the name-to-address translation changes, many users would look at the cached no-longer-valid IP address, and would be unable to reach the host until the TTL expires and a new DNS query is issued. In practice, TTL values are set conservatively: our measurements indicated that periods between changes are considerably longer than respective TTL values. This observation was our underlying motivation for introducing *simultaneous-validation* (SV). Under SV, when a client issues a request to a host (Web server) and a cached expired resolution is available, the proxy/browser issues the HTTP request(s) using the expired address while *simultaneously* issuing a DNS query to resolve the hostname. For transparency and consistency, fetched contents are held and displayed only if the stale address entry is validated by the DNS query results. SV reduces latency since DNS query and communication with the host are performed concurrently rather than sequentially. Our evaluation reveals that the mapping of names to IP-addresses is fairly static, and consequently, estimated SV success rate is over 98%. The SV approach is fundamentally different from preresolving or renewal policies. SV does not impose overhead of additional DNS queries. Deploying SV, however, necessitates caching of expired DNS records and support by both the DNS cache and the browser or proxy server.[6]

Related work

To put in context our proposed enhancement to DNS caching, we contrast it with the more well-studied subject of Web content caching. Content is typically cached beyond its freshness lifetime, but expired entries are validated with the origin before being served. HTTP provides mechanisms for client-driven validation and freshness control [14]. Proposed approaches to reduce user-perceived latency incurred on validation requests included server-driven validations [4], [19], transferring stale cached data to a client while the data validity is being verified [13]. Furthermore, in a recent study we simulated techniques similar to the ones we suggest here for web content caches [8].

Although considerable research targeted validation latency for content caching, it seems that no analogous proposed enhancements were made so far for passive caching of DNS records. DNS caching differs in some basic aspects from content-caching: entries have considerably smaller sizes, storage space is ample with respect to the amount of data, and response sizes are small. Yet, there are also basic resemblances: Query time (cost of a cache miss) is significant and hence cache hit-rate is crucial for reducing perceived-latency; freshness typically expires well before the object is modified; and request sequences (to hostnames for DNS caching and to URLs for content caching) exhibit reference locality and characteristic frequencies.

A recent study by Shaikh *et al* [23] tries to estimate the effect that DNS-based server selection would have on the DNS system. DNS-based server selection reduces the magnitude of TTL values radically (typically smaller than 20 sec). This study shows that these small TTL values are detrimental to latency. It also shows that since clients are often not close to their name servers, DNS based server selection may be far from optimal. Our study shows that proactive DNS caching can efficiently circumvent some of latency associated with these small TTL values.

Wills and Shang [25] also conducted a recent study of DNS lookup costs. They find that DNS lookup time contributed more than one second to approximately 20% of retrievals for Web objects linked by home pages of popular servers. Based on simulations of proxy logs they report that about 6% to 10% of the HTTP requests in the log generate DNS misses. This miss rate go up to 20–30% when they assume that repeated requests to the same server within 15 seconds

[6] An additional minor overhead is a request sent to an old IP address in each of the 2% of the requests when SV fails.

window are carried over the same persistent connection. They also measured low change rates in mappings of hosts to IP addresses. These reported statistics are similar to our measurements reported later in the paper.

Another recent study of DNS performance is by Jung *et al* [17]. This study is based on packet traces of both DNS and TCP traffic in contrast with ours and the studies cited previously which are driven from simulations of proxy traces. Working with real DNS data Jung *et al* were able to discover that about 13% of DNS lookups result in answer that indicated an error. They also observe heavy tail behavior of DNS lookup time, and report that roughly 15% of all lookups require a query packet sent to the root server. By correlating TCP connections and DNS resolutions they deduce that the DNS miss rate is 20–30%. (Their definition of miss was somewhat different from ours, if the address was not cached but a relevant name server was cached they classified the request as a hit.) They also find that most misses are for servers with small TTLs ($<$ 10 mins) as reducing all TTLs to 10 minutes did not affect miss rate too much in their simulations. Working with packet traces Jung *at el* had the advantage of inspecting all DNS traffic, not only traffic related to Web accesses. Our measurements in contrast are for Web related DNS traffic.

These recent studies done independently of ours, emphasis that DNS is a significant latency factor. Its contribution to latency grows with the increase usage of DNS based server selection. Our suggested techniques address this performance bottleneck at a reasonable cost.

Overview

Section II provides background material on the domain name system and some relevant statistics. In Section III we provide and discuss measurements which demonstrating the effect of DNS query time on user-perceived latency. The bulk of our contribution is contained in Sections IV–VI. In Section IV and V we present and evaluate several renewal policies. In Section VI we introduce and evaluate simultaneous-validation. We conclude in Section VII and outline future research issues.

II. The Domain Name System

The current specification of DNS is detailed in RFCs 1034 and 1035 [21], [22] from 1987. Even though designed prior to the onset of the World Wide Web, DNS scaled well with the explosive growth of the domain name space. The currently most widely-used implementation of DNS is the Berkeley Internet Name Domain (BIND) maintained by the Internet Software Consortium [16]. BIND is included as a standard part of most vendors' UNIX offerings.

In graph-theoretic terms, the domain name space is a tree structure, where each node has a label. The *domain* or *domain name* of a node is the list of the labels on the path from the node to the root of the tree, separated with dots. If a node x is a descendant of y we say that x is in the domain of y or is a *subdomain* of y. Each node has information associated with it that is represented by *resource records* (RRs). RRs have different types. The types which are most relevant to resolution of names of Web hosts are RRs of type A, each containing a host address, and RRs of type NS, each containing authoritative name-server for a domain. This set of RRs comprises the domain database. The database is divided up into *zones*, which are distributed among *name-servers*.

A given zone data is typically available from several name-servers to increase resilience to host or link failures. A given name-server typically supports one or more zones. The domain name space is divided into zones by making "cuts" between adjacent nodes of the domain-name tree. (if a cut is between node x to its child y we say that x *delegates* y.) After cutting, each connected piece of the tree constitutes a separate zone. The *authoritative* data for a zone consists of all the RRs attached to all nodes from the top node of the zone down to cuts or leaf nodes. The zone also contains information about delegations of its subzones. This information consists of NS RRs that name the servers for the subzones. In order to process a query, each parent zone must have all the information necessary to access name-servers for their children zones. As we mentioned, the parent zone contains NS RRs that name the name-servers for the subzones. If the name-server itself is not within the subzone, the name of the name-server can be resolved to find an address RR for it. If the name-server is within the subzone, the parent zone must contain *glue* address RR for the name-server. The *local data* of a name-server consists of all the RRs in the zone it is authoritative for, the NS RRs and glue address RRs for the subzones, and the RRs in the cache.

There are two types of queries one can send to a name-server, *recursive* and *iterative*. When processing an *iterative* query, the name-server simply returns the best answer available in its local data. The queried name-server consults its local data, looking for the requested data. If the requested data is not present locally, it returns a referral to a name-server (either a name or a name and an address) serving a zone as far down the domain name tree towards the requested name as it has in its local data. If the query is *recursive* (and the queried name-server supports recursion), then if the answer is not available in its local data it queries the "closest" name-server it has (via either recursive or iterative queries). Hence, the answer to a recursive query either contains the requested data or an error stating that the data requested does not exist. A name-server that does not support recursive queries (such as a root server) performs less work per query but since it never sends out queries, it cannot build up a local cache containing more data. Therefore, name-servers usually send *iterative queries*. On the other hand, the operating system library routines (often termed *resolvers*) interfacing between applications and DNS, usually issue recursive queries to the local name-server.

We illustrate the process through the following example. This example is also illustrated in Figure 1. Suppose we want to browse the page whose URL is `http://www.comnet.technion.ac.il/infocom2000/`. Our browser is linked with an operating system routine called *resolver*. The resolver issues a recursive DNS query to our local name-server `ns.research.att.com`, asking for the IP-address of the host `www.comnet.technion.ac.il` that stores the page. Throughout this example we assume that all caches of queried name-servers do not contain any additional useful information. Our local name-server searches its cache for `www.comnet.technion.ac.il` and when this search fails, it tries to look for an NS RR for one of the ancestor domains `www.comnet.technion.ac.il`, `comnet.technion.ac.il`, `technion.ac.il`, `ac.il`, and `il` in this order. Since we assume that the cache is empty, our name server will only find NS RRs for the root domain. Each such RR contains the name and the address of a name-server authoritative for the root domain, for example `A.ROOT-SERVERS.NET`. Next our local name server `ns.research.att.com` issues an iterative query asking `A.ROOT-SERVERS.NET` for the address of `www.comnet.technion.ac.il`. The root server returns a referral to name-servers for the domain `il` one of which is `RIP.PSG.COM`. Next our local server resolves `RIP.PSG.COM` and once it has the address, it queries `RIP.PSG.COM` for the address of `comnet.technion.ac.il`. The name server `RIP.PSG.COM` returns a referral to `ns.technion.ac.il` that is authoritative for `technion.ac.il` and a glue address RR for it. Last our local server queries `ns.technion.ac.il` for `comnet.technion.ac.il` and gets back the answer.

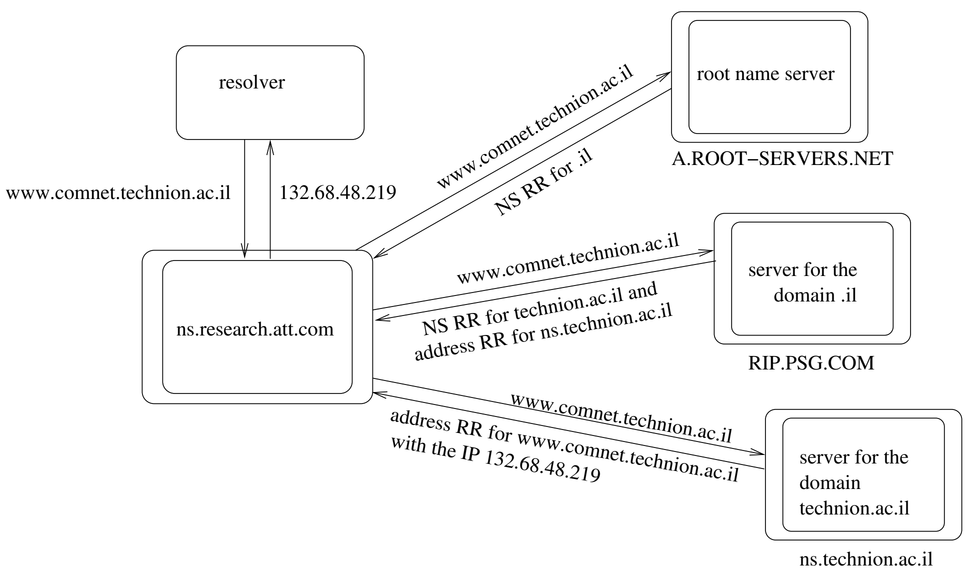


Fig. 1. Illustration of a query for the host www.comnet.technion.ac.il by a browser in the domain research.att.com. This query triggered 3 round trips outside the AT&T network not including possible round trips for finding the address of the name server RIP.PSG.COM.

The number of name-servers contacted to resolve a particular name depends on the contents of the cache in our local name-server and in servers we communicate with through the resolution process. The maximum possible such number equals the number of subdomain delegations on the path from the root to the node representing our queried name in the domain-name tree, and, if name-servers are not located in subzones, additional queries to find addresses of authoritative name servers for the delegated subdomains. Our measurements indicate that the number of delegations on the path to a typical host is often small. As an upper bound we measured the length of the path from the root of the domain name tree to the nearest delegated ancestor of the name. Our statistics based on large proxy logs shows that the length of this path for about 80–90% of the servers is two. Jung *at el* find that among DNS lookups that contacted a remote server, 80% were resolved without any further referrals, which means they got the answer from the local data of the server first contacted.[7] Most remaining lookups (except 1–2%) required contacting one additional server. When the authoritative name server and the host address both are not cached a typical resolution will query the root server which is also authoritative for the `.com` `.org` and `.net` domains. The root server will return a referral to the name-server authoritative for the name. If the returned name-server lies outside the root-server's zone, we would have to resolve it separately. Finally we query the authoritative name-server itself.

Associated with each RR is a time-to-live (TTL) parameter that states how long it can be cached before it should be discarded. The TTL value determines a tradeoff between the load on the authoritative name-server for the RR and how current are copies of the RR in caches of other name-servers. With a longer TTL value, fewer queries are issued to the name-server but copies of the RR in caches throughout the network are less current. When an IP-address of a RR is changed, other hosts may not be able to reach the host for the TTL's duration. This is because a name-server caching the old RR just before the change would use an incorrect address until the TTL expires. BIND allows associating smaller TTLs with individual RRs where a change is expected shortly. Domain administrators, however, seem to simply use relatively small TTL values overall (see Section VI). Figure 2 shows the cumulative distribution of TTLs from various types of RRs. We obtained these TTLS by resolving each server name in our proxy logs until obtaining an authoritative answer. About 25% of the hosts have TTL no greater than an hour and about 90% of them have TTL values no greater than a day. We notice that TTL values of NS RRs and address RRs of name servers tend to be longer than TTLs of the address RRs of the web hosts (Jung *at el* [17] report similar statistics). This reduces the fraction of lookups that trigger access to a root server. Nevertheless even 80% of address RRs of name-servers have relatively short TTL values of no more than a day. Notice also that our ideas of automatically refreshing RR's applies to all kinds of RRs.

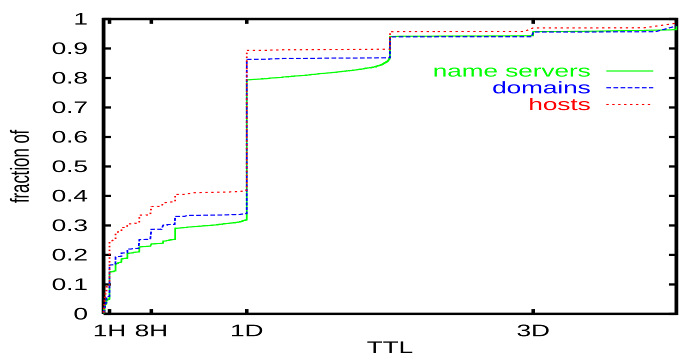


Fig. 2. Cumulative distribution of TTL values for three types of RRs: address RRs of hosts, NS RRs, and address RRs for name-servers

A typical name-server serves many individual clients, and in that sense is similar to a proxy server. Even when the particular address RR for a hostname is not in the cache, the cache may contain useful RRs which shorten the resolution process, for example, NS RRs of names of closer name-servers and/or address RRs for the name-servers. The lists of hosts represented in our logs show that a large fraction reside in a subdomain of a first level domain as `.org` or `.com`, specific to them and delegated to an ISP. That means that most hosts can be associated with unique NS RR of their subdomain as indicated by Figure 3. (This figure is based on measurements for 35K hosts sampled from hosts occurring in a week of traffic through the NLANR cache.) Many of these host-specific zones, however, share the name-server of a common ISP. This is indicated by Figure 4 that is based on the same data as Figure 3. This sharing of name-servers boosts the effectiveness of DNS caching.

III. DNS-related User-Perceived Latency

To further motivate our ideas we show in this section that DNS lookup is a considerable latency factor in the perceived latency of the Web user. To that end, we use some of our own measurements from a recent study in which we "replayed" an AT&T proxy log [7], and additional measurements of query times to two well connected servers of a CDN.

While simulating the AT&T proxy log we measured the relative contribution of DNS resolution to the overall latency of each request. Figures 5 and 6 illustrate the contribution of DNS resolution to overall latency of two groups of requests.

[7] This implies that the address of an authoritative name server was cached.

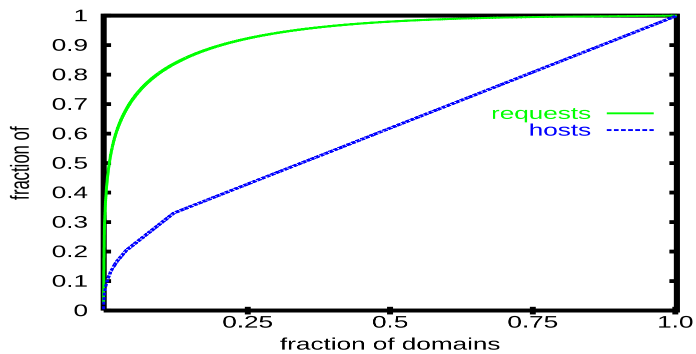


Fig. 3. Fraction of hosts and fraction of requests covered by a given fraction of domains.

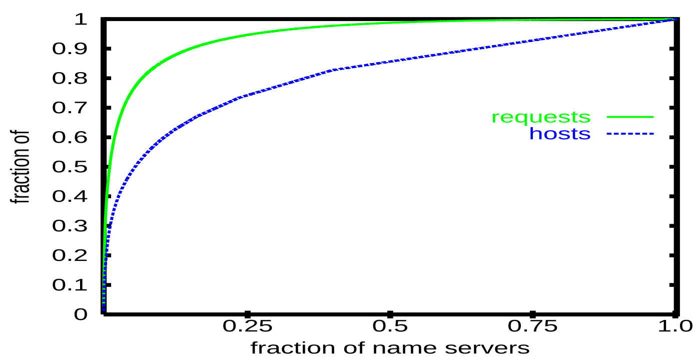


Fig. 4. Fraction of hosts and fraction of requests covered by a given fraction of name-servers.

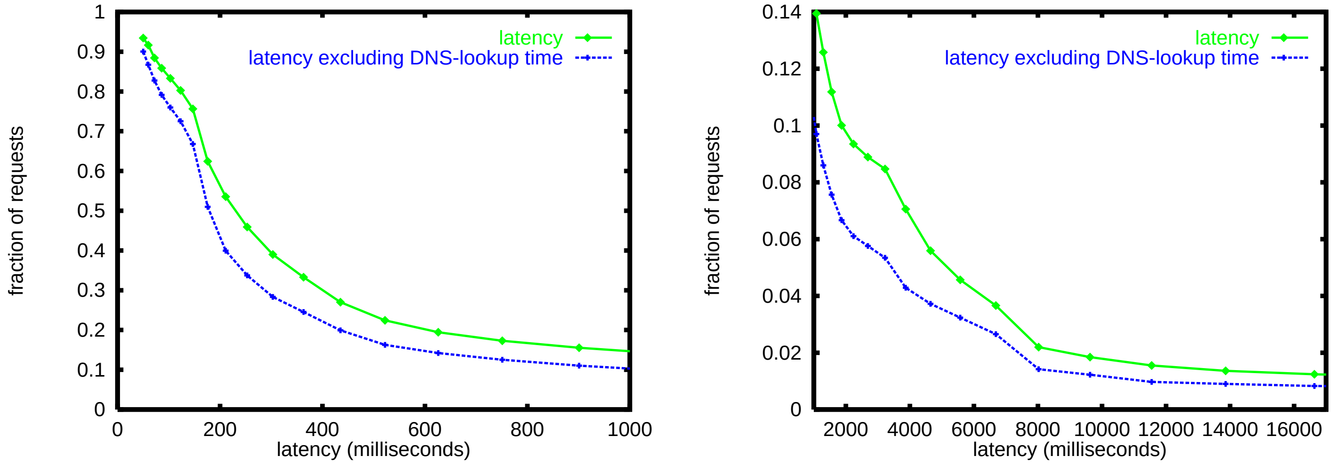
These figures plot the fraction of requests with latency exceeding a certain value (x-axis), including and excluding resolution time. The overall latency of an HTTP GET request was measured as the time to fetch the homepage of the referenced server (usually text without the embedded contents). The simulation assumed browser caching of 15 minutes (equivalently we forced minimum TTL of 15 minutes) and that a single BIND local name-server resolved all and only requests reflected in the trace.

Figure 5 aggregates over all HTTP requests made at least 60 seconds following a previous request to the same hostname. This set of requests attempted to capture "communication setup" time between a user and a Web server. Latency on remaining requests is considerably smaller, since they benefit from persistent connections and pipelining and are not subjected to DNS resolution. When DNS resolution time is excluded, the fraction of such requests with latency exceeding 300ms, 1sec, and 4sec, respectively, drops from 40% to 30%, 14% to 10%, and 7% to 4%, respectively. Figure 6 aggregates over all requests with *referer* field pointing to the AltaVista [2] search engine. This set addresses sites contacted as a result of a search. We suspected that this group of sites are less likely to have been contacted recently and therefore expected it to be more volunerable to long DNS query times. Indeed, t he figure shows a greater relative effect of DNS query times than seen for the first group. When DNS query time is excluded, the fraction of requests with latency exceeding 1sec drops from 35% to 22% and the fraction with latency exceeding 4sec drops from 18% to 10%.

We also observed that DNS causes significant delays even while communicating with name servers that belong to a content delivery network and therefore are expected to be well connected and well maintained. To show that, we focused on the two hostnames `a111.g.akamaitech.net` and `a4.g.akamaitech.net` that have been recorded by the UC cache of the NLANR network from January 20–29, 2000. There were 7351 logged requests to `a111.g.akamaitech.net` and 1759 logged requests to `a4.g.akamaitech.net`.

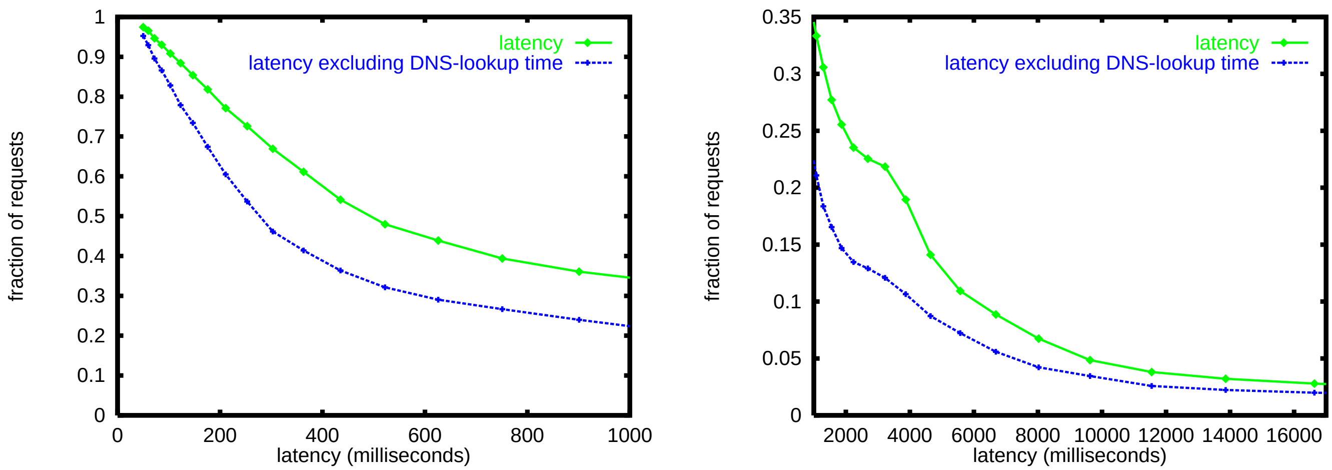
We recorded the TTLs of the RRs involved in the resolution process of these hostnames. The TTL of the NS RR of the first-level domain (`akamaitech.net`) was 90000 seconds (25 hours), and the TTL of the NS RR of the domain `g.akamaitech.net` was 1000 seconds.[8] The TTLs of the address RR of `a111.g.akamaitech.net` and the address RR of `a4.g.akamaitech.net` were 20 seconds. These TTL values imply that a very small fraction of the requests to these two hosts (at most one request a day) involved a name server outside the Akamai domain (in which case

[8]The address records of the corresponding name servers had the same TTLs.



0–1 seconds 1–17 seconds

Fig. 5. Relative effect of DNS lookup time on total latency on all requests issued within 60 seconds or more after the previous request made to the same Web server (AT&T log).



0–1 seconds 1–17 seconds

Fig. 6. Relative effect of DNS lookup time on total latency on AltaVista follow-up requests (AT&T log).

it must be a root server). Furthermore the short TTL of the hosts themselves imply that almost every request triggered a DNS query (if no minimum TTL is practically imposed by the browser).

For these two particular servers, 90% of HTTP requests were served in less than 200 milliseconds, about 4% exceeded 3 seconds, and only about 1% exceeded 5 seconds. We also performed a separate set of DNS lookups for these hostnames from our Lab. We issued queries with 25 second intervals over a 24 hour period. 85% of lookups returned in less than 50 milliseconds, 90% within less than 100 milliseconds, and about 10% of the lookups took longer than 3 seconds. These measurements strongly suggest that DNS is a considerable latency factor even for well connected popular hosts. Wills and Shang [25] also measured DNS lookup time of over a second among 20% of a set of servers linked from home pages of popular sites.

Timeout and retransmission occur when a response is not received within the specified period. This could be due to long RTT, a packet getting discarded by a loaded server, or packet loss at the network. Low rate of packet loss at the network (close to the ISP's name server) could be the explanation for long delays to the Akamai name servers, since their load and location are carefully managed. Packet loss is particularly detrimental for DNS exchanges (one UDP packet each for the request and response), since each lost packet means a timeout-value delay (in the order of seconds) to the end-user. Packet loss is not as detrimental to TCP connections, since longer timeout values are issued only during the handshake (connection establishment), and otherwise the value is set adaptively, typically to considerably shorter values, according to estimated RTT.

IV. RENEWAL POLICIES

Name-servers receive and resolve DNS queries and cache and reuse records for the time period specified in their TTL value.[9] A client query that can be answered from the local cache is labeled *cache hit*. Otherwise, resolving the client query involves issuing queries to remote name server(s) and the client query constitutes a *cache miss*. Under *passive caching*, currently practiced by BIND, DNS queries are issued by the name-server only as a result of a cache miss. In contrast, *proactive caching* uses "unsolicited" queries (*automatic-queries*) in order to increase hit-rate on client queries. On average, more than one automatic-query is performed in order to avoid one cache miss. Our premise, however, is that user-time is valuable and thus a cache miss is "costlier" than a respective automatic query.

A. Cost Model

We associate costs with both automatic-queries and cache misses. The cost of an automatic-query attempts to price the overhead imposed on name-servers and the network. Cache misses cost correspond to the associated user-perceived latency. Our *basic cost model* counts the number of automatic queries vs. the number of cache misses. *Refined cost measures* account for varying query-complexity (number of different name-servers contacted) for measuring automatic-query cost and for elapsed query times to measure miss cost. We use the basic cost model in Section V-D and address the refined cost measures in Sections V-E and V-F. For each proposed policy we consider the tradeoff that correspond to different ratios between *miss cost* and *automatic-queries cost*. Overhead and latency costs are dependent, as significant increase in traffic to name servers would result in increased latency. We chose, however, to separate the two since DNS traffic even if increased by a small constant factor would still constitute a small fraction of the overall traffic and should not strongly affect user response time. We also expect that even if this dependence is factored in the relative performance of different policies would stay the same.

This model assumes that storage space is plentiful and records do not have to be evicted before they expire. This is consistent with our data since only about 200K distinct servers were seen by the 3 large NLANR caches combined (see Table I) over a period of two weeks. The AT&T proxy trace included only 13K distinct servers. The combined size of all associated RRs can easily fit on a small part of a hard disk (or even in memory of a dedicated PC). This assumption of abundant storage is also implicit in (the UNIX version of) BIND, that allows its cache to grow until maximum process-size is exceeded and the application is killed [1].

The *passive policy* corresponds to a single point on the performance curve. It performs a single query for each miss. One can easily establish by induction that the passive policy performs the minimum possible number of DNS queries needed to serve all client requests.

The optimal proactive caching policy is the omniscient OPT that uses knowledge of the future: An automatic-query is issued by the name-server *just before* it receives each client-query that would otherwise be a miss. OPT incurs no overhead (issues the minimum number of queries needed to serve the sequence of client-requests)[10] and suffers no cache misses. In the following, we restrict our attention to proactive caching policies that are only allowed to extend freshness of cached items by renewing items as they expire. These policies exclude predictive renewal of long-expired items or prefetching of new ones. We label such policies *renewal policies*.

B. Policies

All the policies that we considered are *renewal policies* which are defined as follows.

- A *renewal* of a cached entry is performing a new resolution upon its expiration, and updating the cached copy and extending expiration time accordingly.
- A *renewal policy* associates with each cached item a *renewal credit*, which is an integer stating the remaining number of renewals. The credit may be updated (increased) when the cached item is used. When the item expires (or is about to expire) and has a positive credit, it is renewed and the credit is decremented.

An analogy between renewal policies and cache replacement policies can be seen through the cost measures: cache replacement policies are measured by the tradeoffs of *cache-storage vs. hits*. For renewals, items are charged "rent" (renewal cost) for staying available (cached with non-expired TTLs). The underlying performance tradeoff is that of

[9] We limit the discussion in this section to data not associated with the local zone.

[10] We assume that under the passive policy, requests do not occur at the exact end of a TTL interval.

"total rent" vs. hits. The operational mode is also similar since both do not prefetch new items according to "external" hints. Instead, they value cached items by properties of the request sequence such as locality and per-item popularity.

We describe our renewal policies by specifying for each how it assigns renewal credits. Our policy-design principles were to only use information available locally at the name-server and to be at least as simple to implement as popular cache-replacement policies. We named our renewal policies after analogous cache-replacement policies, where analogies were made based on the property of the request sequence exploited. Policies are parameterized and cost-benefit tradeoffs are obtained by sweeping the parameter value.

- R-FIFO(r): A fixed number of renewals, r, is associated with each item upon a cache miss. The renewal credit is assigned at the point of entry into the cache and is not increased by subsequent hits. This is analogous to the cache replacement policy FIFO (First-In-First-Out) that upon a miss evicts the item with the earliest entry-time into the cache.
- R-LRU(r): Each item is renewed for r times *passed the time of the most-recent cache-hit involving the item*. Hence, when a cache-hit involves an item, its renewal credit is set to r. This is analogous to LRU (Least Recently Used) that evicts the item with the least-recent cache hit.
- R-LRU(ϵ, r) is similar to R-LRU(r), but the renewal credit is reset to r only as a result of hits that occur after an ϵ-fraction of the "current" TTL interval had passed.

Our motivation for considering R-LRU(ϵ, r) is that plain R-LRU always grants at least 1 renewal when the miss is followed by at least one consecutive hit. R-LRU(ϵ, r) avoids an extra renewal if there is no "deeper" evidence of need such as later hits stemming from requests initiated at a different session.

- R-LFU(r): The first request to the item *in each TTL interval* increases the renewal credit by r. R-LFU(r) is analogous to LFU (Least Frequently Used) that evicts the cached item with smallest number of hits. Both LFU and R-LFU give additional value to an item for every cache hit.
- R-ADAPTIVE: is an enhancement of R-LRU where different credit $r(h)$, is associated with each host h. To find the mapping $r(h)$, R-ADAPTIVE collects per-host statistics from "learning data." Each host h that had sufficiently many misses under R-LRU with $r = 1$ on the "learning data," gets a value $r(h) \geq 1$. A generic value $r(h) \equiv r \geq 1$ is used for hosts for which there was no sufficient data. The mapping of hosts to values $r(h)$ is performed using similar techniques as the ones used in [10], [9] and analyzed in [6]. The implementation of R-ADAPTIVE is considerably more involved than R-LRU, R-FIFO, or R-LFU and requires generating and maintaining the mapping $r(h)$.
- R-OPT: is the optimal renewal policy. The performance curve of R-OPT gives for any number of renewals, the minimum possible number of cache misses. R-OPT relies on knowing the future. Given the request sequence as an (offline) input, the optimal tradeoff-curve can be computed using dynamic programming. [11] For our experiments we implemented R-OPT(r), a simpler greedy approximation of R-OPT. The greedy approximation performs a single pass on the input and grants renewals only if the gap between the expiration and the next request is less than r TTLs. The greedy approximation can be viewed as analogous to Belady's cache replacement algorithm [3]. The approximation collapses to the optimal solution if renewals can be granted for fractional TTL values.

Note that R-OPT differs from the optimal proactive caching policy outlined above since it follows the more restrictive framework of renewal policies. As such, it provides a more realistic tighter performance upper-bounds.

Varying renewal and miss costs can be naturally integrated into our policies definitions. Our initial evaluation uses the basic model and we subsequently adjust by incorporate projected query times.

V. Performance Evaluation of Policies

A. Data

Our experiments extrapolated workload and performance in name-server caches from proxy logs. We used logs from 3 of the large NLANR Web caches (downloaded from the NLANR site [15]) and a proxy log from the AT&T Research

[11] To see this dynamic programming scheme consider each hostname separately. Let $r_1, \ldots, r_n$ be the sequence of requests to resolve one hostname. Let $c(i, j)$ be the "best way" to cover $r_1, \ldots, r_j$ with $i \leq j$ misses. That is among the coverings that minimize the number of queries, we consider the one that extends as much time beyond r_j as possible. Note that $c(i, j)$ is not defined for all values. The maximum i for which $c(i, k)$ is defined corresponds to the passive policy. The range of values $c(i, k)$ (minimum number of queries for covering the complete sequence with i misses) corresponds to a tradeoff curve of queries vs. misses. For the optimal policy we actually focus on additional queries vs. reduction in misses with respect to passive. For each hostname, the quantities $c(i, j)$ can be computed using dynamic programming from $c(i', j')$ such that $(j', i') < (j, i)$ in lexicographic order. To obtain a globally optimal solution, we use a threshold value V to select the number of renewals used for each hostname in an "equal way." The "fractional knapsack" strategy outlined in [6] is applicable.

label	requests (thousands)	end hosts (thousands)	time-period
LJ	4103	63	May 18–June 5/99
UC	10837	91	May 18–June 5/99
PA	6886	104	May 18–June 5/99
AT&T	489	10.5	Nov 8–19/96

TABLE I
PROXY LOGS

proxy. The NLANR caches are high-volume, with large rate of requests and clients that include many proxy caches, whereas the AT&T proxy log reflects the activity of about 460 individual users (IP-addresses). Properties of the different logs are provided in Table I. We considered only requests made to hosts logged by name that we were able to resolve successfully. This was the vast majority for the NLANR logs, but included only 10.5K out of 13K servers for the older AT&T proxy log. The NLANR logs were split into two parts in order to obtain a learning and test data for evaluating the R-ADAPTIVE policy. Learning part of each log included the first 8 days, and the second part the remaining 11 days.

The proxy logs include information about (anonymized) user identity,[12] Web server (host), requested resource (URL), and cache performance (whether the request was a local hit, obtained from another NLANR cache, or obtained from the Web server). The logs, however, do not include any information on DNS queries and results. DNS data is crucial for our performance evaluation and was projected from independent measurements.

B. *Obtaining DNS Data*

We extracted the list of distinct hosts from the logs, and resolved them using `DIG` [1]. Thus, we obtained information on associated IP addresses, aliases (canonical names), immediate name-servers, and DNS query times, along with respective TTL values. We issued several queries for each host name, with varying time intervals between them, in order to estimate rate-of-change of name-to-address mappings, query times, and TTL values. We used these TTL values to emulate the local name-server cache and its performance under various policies. The name-to-address rate-of-change was used for evaluating the performance of SV (see Section VI) We performed these measurement few weeks after the NLANR logs were recorded.

C. *Methodology*

When projecting on the activity of the local name-server, we assumed in our evaluation that it exclusively handles DNS queries associated with all requests in the proxy log. The number of resolutions, however, is considerably smaller than the number of logged HTTP requests. With persistent HTTP connections (incorporated in HTTP/1.1 [14]), requests for embedded contents and requests issued shortly after a previous request to the same host re-use an existing TCP connection, and implicitly, re-use DNS resolution results. Even though some RR have very small TTL values (even 0), (persistent) TCP connections to a host are not aware of expiration of the address RR, and are not terminated when the latter expires. To simulate name-servers co-located at a proxy, we assumed that no resolutions are performed on requests occurring less than 60 seconds after a previous request to the same host. (Notice that Web browsers may impose larger minimum TTL values when caching name-to-address entries.) We performed sensitivity analysis by varying minimum effective TTL between 0 and 15 minutes: The DNS cache miss-rate decreased by about 20% (since many projected misses are due to small TTLs), but our results did not qualitatively change: The reduction in the miss-rate for a given cost, achieved by each policy was the same.

D. *Performance in the basic cost model*

For each policy we measure the tradeoff between the relative decrease in cache misses and the relative increase in DNS queries (issued by the name-server). As the natural baseline we use the passive policy, whose performance on the various logs is listed in Table II. The table provides, for each log, the total number of DNS misses, the number of DNS

[12] For the NLANR caches a different anonymization was used every day.

log	DNS misses (% HTTP requests)	FS misses (% DNS misses)
LJ	337964 (8.2 %)	63199 (18.7%)
UC	941150 (8.7 %)	91291 (9.7%)
PA	592278 (8.6%)	104240 (17.6%)
AT&T	31732 (6.5%)	10471 (33.0%)

TABLE II
PERFORMANCE OF PASSIVE CACHING

misses divided by the number of HTTP requests, and the fraction of DNS misses that are associated with the first HTTP request to a hostname. The number of queries issued by the passive policy is equal to the number of DNS misses.

Notice that we consider requests occurring within our imposed minimum TTL interval of 60 seconds as hits and as a result our miss rates are relatively low. Alternatively, one could define the miss rates as the number of misses divided by the total number of requests excluding ones within an imposed minimum TTL values (i.e. excluding requests likely to be carried over the same persistent TCP connection). As reported by Wills and Shang [25] for a minimum TTL of 15 seconds that would increase miss rates to around 25%. This number is closer although probably still smaller than the actual miss rate at the name-server's cache, since as estimated by Shaikh *at el* [23] approximately one in eight HTTP requests generates a DNS query. Since we are mainly interested in the relative decrease in miss-rate that is possible for a given cost, and this decrease was not sensitive to the imposed minimum TTL value, the particular choice of the definition of miss rate is not important (we could have used actual the miss counts). However, the reader should not be mislead by the low numbers to underestimate the possible benefit from our techniques. The low miss rate are just an artifact of our particular definition of miss rate.

We refer to DNS misses that are associated with previously-seen hosts as *PS misses* and to misses incurred on the first appearance of a hostname as *FS misses*. The number of PS misses constitutes an upper bound on the performance of renewal policies, since only such misses can be eliminated. It is meaningful to consider also performance in terms of PS misses, since it is robust to varying log duration. The fraction of PS misses increases and converges with log duration (e.g., 150K distinct host seen on the first 8 days of the 3 NLANR caches whereas the following 6 days included only 50K additional hosts).

Figure 7 plots the performance of the various policies on the AT&T proxy log and the second part of the UC log. The relative performance of the different policies was consistent across all 4 logs and the two parts of each log. R-FIFO consistently under-performed other policies. The performance of R-LRU, R-LFU, R-LRU(ϵ =0.1) and R-ADAPTIVE was comparable. R-LRU(ϵ =0.1) and R-ADAPTIVE provided only minor performance improvements over plain R-LRU. The perhaps smaller-than-expected advantage of R-ADAPTIVE is explained by a large number of hosts with a small number of "misses" on the learning data. Thus, R-ADAPTIVE associated the generic renewal credit amount with these hosts. These results also imply that the added complexity of implementing R-ADAPTIVE is not justified by the performance gain.

Figure 8 plots the performance of R-LFU on all 4 logs. The tradeoffs obtained by R-LRU closely follow R-LFU. The convexity of the curves implies that the relative decrease in the number of misses becomes smaller as we grant more renewals. Notice that as higher the number of renewals, items stay longer in the cache, and we eliminate misses even if the interrequest interval is longer. So the convexity of the curves further suggests that cache misses were more likely to occur on popular and more recently-requested objects.

We also observe that better tradeoffs were obtained for the larger logs. This is due to both the fraction of PS misses that increases with log period and to larger number of clients. When the fraction of PS misses is factored out (by measuring reduction in PS misses instead of reduction in the total number of misses), the tradeoffs obtained for different logs move closer together. Across the 3 NLANR logs the tradeoffs completely converged, but the AT&T log that has a considerably-smaller client-base still exhibits worse tradeoffs. On the NLANR logs, R-LRU and R-LFU eliminate about 60% of PS-misses with query overhead of 2 and eliminate 80% of PS-misses with query overhead of 5. The respective reductions for the AT&T log are 36% and 63%.

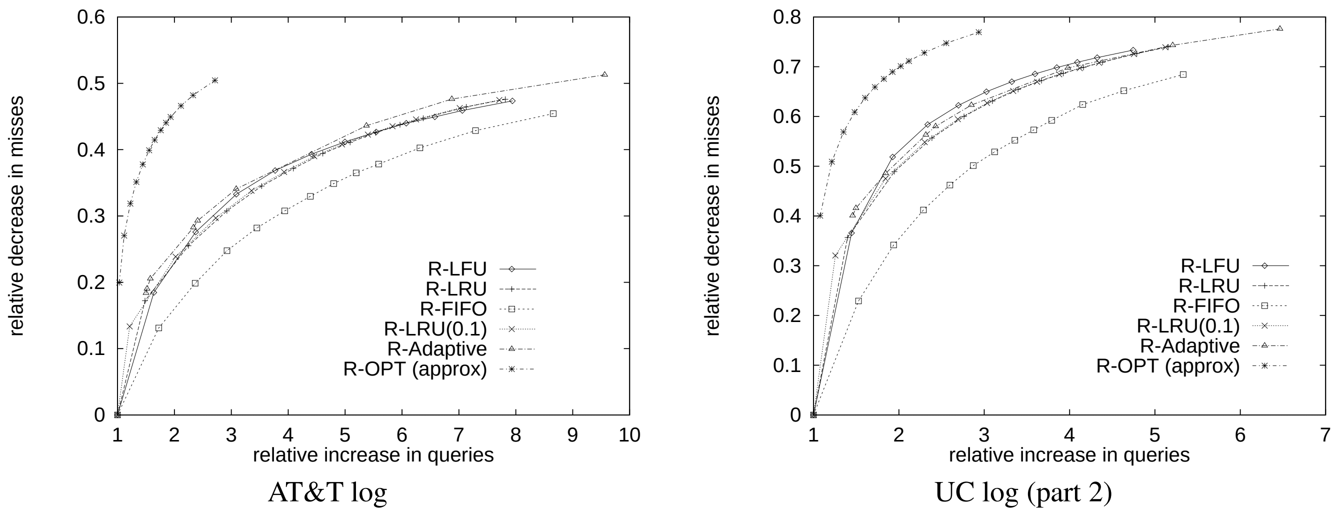


Fig. 7. Performance of renewal policies. By varying the parameter r of the policies we obtained these tradeoff curves. (Each value of r for a particular policy gives us a single point on its curve.) As r grows we generate more renewals. The x-axis gives the increase in the number of DNS queries due to these renewals.)

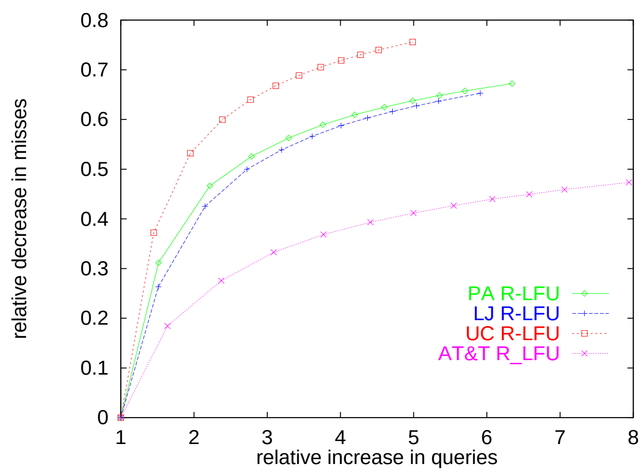


Fig. 8. Performance of R-LFU on the 4 logs

E. Renewal costs

Renewal costs is measured by the overhead inflicted on name-servers throughout the distributed DNS system. It is hard to capture precisely, but can be estimated by the number of different subqueries issued in the recursive resolution process, or by considering the number of queries issued to the loaded root DNS servers (see Section II).

Although many DNS queries involve two subqueries, one to the root server and another to an authoritative server for the target, renewals are more likely to involve only a single subquery to a lower-level server. We considered TTL values for hosts and their immediate name-servers and noticed that TTL values for both immediate name-servers and their address records are larger or equal to the host TTL for 89% of hostnames in the AT&T log and for 87% of hostnames in the first week of the LJ log. This suggests performing renewals *just before* the expiration of the host address record, by directly contacting (the generally still available) immediate name-server. Such *pre-expiration* renewals are resolved by a single subquery and guarantee an authoritative response with the maximum TTL value. This subquery often renews the name-server record itself, allowing for continued pre-expiration benefits in subsequent renewals. Thus renewals, and in particular pre-expiration renewals, are more likely to involve only lower-level lightly-loaded name servers.

F. Integrating query times

Query times associated with cache misses may vary substantially and depend on the hostname and cache content (see Section II). Thus, when evaluating performance, it is important to relate the reduction in the number of misses with the reduction in the number of misses *with long query times*. Since respective DNS query times were not provided with the traces, we project query times using separate measurements. The cache is primed by previous requests to the same hostname and hostnames sharing higher-level domains, hence the elapsed time since a previous request should be accounted for when projecting resolutions times. We issued DNS queries to hostnames in the AT&T log with varying time intervals between them. We associated them with DNS misses in our simulation according to elapsed time since previous cache-miss involving the same hostname. Figure 9 shows the number of DNS misses with (projected) query times exceeding $x \geq 500$ms. The upper-most curve corresponds to misses incurred by the passive policy. There were 31.7K misses in total (see Table II), and the figure shows that about 1.6K misses (5%) had query times exceeding 3 seconds. The lower-most curve aggregates across all FS misses. This is the baseline curve, since renewal policies (and SV) apply only to PS misses. The curve for FS misses is halfway of the passive-policy curve throughout query times, showing that FS misses account for about 1/2 of the misses with long query times (over 500ms or more). Recall that FS misses account for about 1/3 of the total number of misses of the passive policy on the AT&T log (see Table II). This gap arises because FS misses are incurred on hostnames that were not previously resolved, and hence, are more likely to incur longer query times. The figure also shows the latency-distribution for misses incurred under R-LFU with different values of r and respective renewal overhead. The number of long query times does not decrease proportionally to the total number of misses. This is expected, since renewal policies target misses occurring sooner after a previous resolution. The figures also show, however, that the number of long query times still significantly decreases as the total number of misses decreases. R-LFU with $r = 2$, for example, performs 140% more queries than passive and eliminates 40% of the long query times incurred on PS misses. R-LRU (not shown) exhibits comparable performance. Last, the figure shows query times under SV, which is discussed in Section VI.

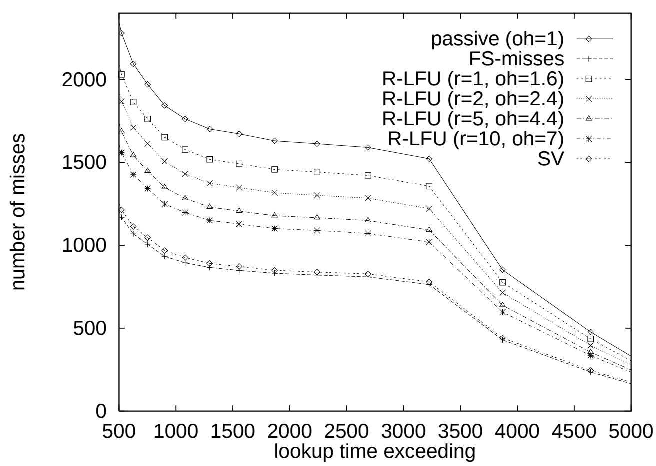


Fig. 9. Number of misses with long query times

G. Implementation guidelines

Most effectively, renewal policies can be integrated in the name-server software. The implementation can then use the cache content and query history and benefit from performing pre-expiration renewals. Alternative implementation is as a process external to the name-server that issues queries to the name-server. Incorporating a renewal policy in the name-server software requires both (1) maintaining and updating "remaining number of renewals" and (2) scheduling renewals. The policies considered here update the number of remaining renewals only when the item is fetched, is involved in a cache hit, or is being renewed. Thus, updates happen only when a RR is accessed and if the remaining number of renewals is stored together with each record, the update overhead is minimal. The scheduling and issuing of automatic-queries can be performed through periodic scans of the cache. Periodic scans for removing expired items are incorporated in BIND 8. Different implementation that allows for frequent renewals is via a *priority queue* [11]. Scheduling renewals in a queue allows for prompt and quick identification of currently-due items while avoiding frequent complete scans of the data. When TTL lengths allow for sufficient flexibility, it may be desirable to perform renewals at off-peak times.

VI. Simultaneous Validation

Simultaneous validation (SV) reduces total document-fetching time by *concurrently* performing the hostname resolutions and subsequent parts of the process (TCP connection establishment and HTTP request-response). It is potentially effective when a fresh entry for the hostname is not available at the local cache, but the cache contains an expired entry obtained from a previous resolution or from elsewhere. SV needs to be supported at the entity initiating TCP connections to Web servers, typically, browsers or proxy servers. With SV deployed, expired DNS records are not discarded, but kept cached for SV use. When a client issues an HTTP request, SV proceeds as follows:

1. If a fresh address record is available, or if no (fresh or expired) address record is available, the process proceeds as if SV is not deployed.
2. Otherwise, if an expired address record is locally available (from browser cache or from a local SV-supporting name-server cache), then concurrently:
 (a) the expired address record is used to attempt a TCP connection establishment.[13] If a connection was established, HTTP request(s) are issued and responses are stored locally.
 (b) The resolver performs a DNS query of the host name. The query response (and subquery responses) update and replaces respective cached records.

SV may continue in several manners, depending on the desired approach. A *conservative* implementation guarantees that the deployment of SV is completely transparent to the user's view (other than reduced latency). Contents fetched in the first step are cached and transferred to the browser (if deployment is at proxy) or displayed to the user (for Web browser deployment) only if the IP-address is validated by the results of the simultaneous DNS query. The latency experienced by the user under a conservative implementation is the maximum (rather than the sum) of DNS query time and the remaining portion of the process serving the request.

Under an *aggressive* implementation of SV, fetched contents are displayed before DNS query response is received. An aggressive implementation is beneficial when DNS query times extend longer than the rest of the process or when relevant name-servers are temporarily unavailable. In these cases, perceived-latency is reduced by DNS-query time. The main disadvantage of an aggressive implementation is compromising transparency. When query response does not validate the expired address, it may result in serving the user with stale or unintended contents. If validation failed, the aggressive implementation has the following options: 1) consider the displayed results as valid; 2) validate the fetched content [14] by contacting the host using a valid address; 3) fetch the content from a valid address and re-display if there is discrepancy.

A. Rate of Change of IP-addresses

We estimated the rate-of-change of hostname to IP-address mappings. We extracted the list of hostnames from the logs and resolved these names repeatedly with time intervals varying from 10 minutes to 10 days. To estimate change intervals, we took the first IP-address in each resolution of a host-name and checked whether it is present in the list of addresses provided in each of the subsequent resolutions. When the address was not included in the resolution results, we considered it as an address change. Our measurements showed a low rate-of-change for the groups of hosts in the NLANR and AT&T logs, where only 2–3% of hosts had any address changes. A small subset of hosts, including many image and advertising servers, exhibited very frequent changes. About 0.25–0.5% changed daily and fewer than 0.1% changed hourly.

B. Performance Evaluation

We evaluated the performance of SV using trace-based simulations with the data and general methodology as outlined in Section IV. Our simulation assumed underlying passive caching at the DNS cache. That is, that cached records are obtained only as a consequence of serving client queries. We used the estimated rate-of-change measurements to associate address changes with DNS misses.

When implemented over a passive cache, SV is applicable only to PS misses (see Table II for the fraction of DNS misses that are PS-misses). Hence, we measure the performance of SV by considering all PS misses and measuring the fraction of such misses that were *SV hits*, that is SV was successful (the address did not change since the time of the

[13] If the record includes an address of more than one mirror, we can attempt to establish connection to each mirror, in the same way that we would have used a valid record.

log	SV success rate (% PS misses)
LJ	98.8%
PA	99.1%
UC	99.1%
AT&T	97.9%

TABLE III
PERFORMANCE OF CONSERVATIVE SV ON VARIOUS LOGS: PERCENTAGE OF SV HITS OUT OF PS-MISSES

previous resolution). Performance results for SV on the 4 logs are provided in Table III showing success rate of 98–99%. Note that our evaluation correspond to a conservative implementation of SV, and an aggressive implementation may yield an even higher success rate. Figure 9 shows the corresponding projected query times for SV misses (including FS misses). As we can see, SV is also highly effective on PS misses with long query times. The gap between the SV curve and the FS-misses curve in Figure 9 represents the 2% of PS-misses that SV does not eliminate.

C. Implementation

Simultaneous-validation requires a two-fold resolution of host names: locally-available expired address records are returned quickly and a resolution is initiated in the standard manner to obtain current addresses. SV can be supported by stand-alone implementation at browsers or proxy-servers. Popular Web browsers today cache name-to-address translations, and used fixed-timeout or fixed-size name cache that disregards TTL values. Browsers can implement SV by long-term caching of name-to-address resolutions. SV can then complement existing implementation. If TTL values are supported by the browser, they would determine expiration times. Otherwise, the same fixed-timeout mechanism or other mechanism can be used to determine expiration. Similarly, proxy servers can implement SV by maintaining a local cache of name-to-address records that is separate from the name-server. Implementation at proxy-servers benefits from aggregation across multiple users, with more up-to-date and extensive caches.

Stand-alone implementations at browsers or proxy benefits from modifying only a single entity. Nevertheless, browsers or proxies can still benefit from SV support by the local name-server. Name-servers are the natural provider of two-fold resolutions: they already handle DNS requests from multiple users and cache DNS records. To support SV, they need to (i) cache, instead of discard, expired contents[14] and (ii) provide protocol support for two-fold resolutions. Protocol support for SV can be facilitated through new types of queries by which a client indicates interest in possibly expired records. The resolver will use such a query to indicate to its name-server that it is willing to accept stale cache entries. When a stale entry is returned, the name-server should follow up with a valid response.

D. Coherence Support for Centralized Database

A future application of SV, that extends its applicability to cover FS misses, is to provide coherence to an extensive hostname database (e.g., obtained by extracting lists of hostnames from multiple sources). Since the database entries are not guaranteed to be fresh, the extent and reliability of refreshes can be limited. The effectiveness of such a database can be further increased if other attributes are tracked such as frequency of change of IP-addresses and usage of Round-Robin DNS. It can then focus on hostnames with stable addresses which are more likely to be SV hits and also support round-robin rotation of mirrors. It is interesting to contrast this with the historic approach that predated DNS. Before DNS was designed there was a single file called HOSTS.TXT containing all address translations that was priodically ftped by the hosts in the network. HOSTS.TXT made the complete name-to-address database locally availability, and thus, DNS lookup times were minimal. This approach, however, was not scalable because of growing file size and a coherence mechanism which did not allow for distributed control. In retrospect, storage is less of a concern but coherence remains crucial. The combination of SV and the current DNS infrastructure addresses coherence.

[14]Currently BIND 8 periodically discards expired records.

VII. Discussion

Latency incurred on DNS misses is inherent in the hierarchical/distributed nature of DNS and is often dominated by RTTs to multiple destinations. As such, query time is not considerably shortened when bandwidth is increased. Nonetheless, reducing perceived-latency due to query times is crucial for improving the experience of web users. We view enhancement to current passive caching of DNS data as a necessary step in the ultimate goal of reducing Web latency. To this end, we proposed and demonstrated the effectiveness of two orthogonal approaches: renewal policies and simultaneous-validations. Renewal policies are effective for hostnames where the typical time interval between requests is within some small factor of the TTL. SV is effective when frequency of change of hostname to address mapping is lower than the frequency of requests. They also differ in their place of implementation: Renewal policies can be implemented only inside the name-server whereas SV needs to also be supported by the entity issuing HTTP requests. These two different solutions suggest a combined approach.

Renewal policies apply to all kinds of Resource Records (rather than just to address records of hosts). That is, we can consider address-records of name-server, CNAME records, or NS records, as separate items, and renew them. It is better to perform renewals via iterative queries. The motivation is that often the sets of subqueries performed when resolving different hostnames overlap (for example, when the hostnames share the top-level domain or when they use the same name-server). Thus, even resolutions of first-seen hostnames could be shortened.

Acknowledgements

The authors thank the referees for their valuable suggestions.

References

[1] P. Albitz and C. Liu. *DNS and BIND*. O'Reilly, Cambridge, MA, 3 edition, 1998.
[2] AltaVista. `http://www.altavista.com`.
[3] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM systems journal*, 5:78–101, 1966.
[4] P. Cao and C. Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.
[5] E. Cohen and H. Kaplan. Proactive caching of dns records: Addressing a performance bottleneck. In *Proc. Symp. on Applications and the Internet (SAINT)*, pages 85–94, 2001.
[6] E. Cohen and H. Kaplan. Exploiting regularities in web traffic patterns for cache replacement. *Algorithmica*, 33(3):300–334, 2002.
[7] E. Cohen and H. Kaplan. Prefetching the means for document transfer: a new approach for reducing. *Computer Networks*, 39(4):437–455, 2002.
[8] E. Cohen and H. Kaplan. Refreshment policies for web content caches. *Computer Networks*, 38(6):795–808, 2002.
[9] E. Cohen, H. Kaplan, and J. D. Oldham. Policies for managing TCP connections under persistent HTTP. In *Proceedings of the World Wide Web-8 Conference*, pages 1709–1723, 1999.
[10] E. Cohen, B. Krishnamurthy, and J. Rexford. Evaluating server-assisted cache replacement in the Web. In *Proceedings of the 6th European Symposium on Algorithms*, pages 307–319. Springer-Verlag, Lecture Notes in Computer Science Vol. 1461, August 1998.
[11] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. McGraw-Hill Book Co., New York, 1990.
[12] M. Crovella and P. Barford. The network effects of prefetching. In *Proceedings of the IEEE INFOCOM Conference*, pages 1232–1239, 1998.
[13] A. Dingle and T. Partl. Web cache coherence. In *Proceedings of the Fifth International World Wide Web Conference*, pages 907–920, May 1996. Computer Networks and ISDN Systems 28:7-11.
[14] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, june 1999.
[15] A Distributed Testbed for National Information Provisioning. `http://www.ircache.net`.
[16] Internet Software Consortium. `http://www.isc.org`.
[17] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. Dns performance and the effectiveness of caching. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2001.
[18] S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran. An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *J. on selected areas in communication*, 13(8):1371–1382, 1995.
[19] D. Li and D. R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 1–12, 1999.
[20] M. Mikhailov and C. E. Wills. Embedded objects in web pages. Technical Report WPI-CS-TR-00-05, Computer Science Department, Worcester Polytechnic Institute, 2000.
[21] P. Mockapetris. Domain names – concepts and facilities. RFC 1034, ISI, November 1987. `http://www.dns.net/dnsrd/rfc/rfc1034/rfc1034.html`.
[22] P. Mockapetris. Domain names – implementation and specification. RFC 1035, ISI, November 1987. `http://www.dns.net/dnsrd/rfc/rfc1035/rfc1035.html`.

[23] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of dns-based server selection. In *Proceedings IEEE INFOCOM*, pages 1801–1810, 2001.

[24] K. Thompson, G. Miller, and M. Wilder. Wide-area internet traffic patterns and charateristics. *IEEE Network*, 11(6):10–23, 1997.

[25] C. E. Wills and H. Shang. The contribution of dns lookup costs to web object retrieval. Technical Report WPI-CS-TR-00-12, Computer Science Department, Worcester Polytechnic Institute, 2000.