Intelligent Caching for World-Wide Web Objects

Duane Wessels
University of Colorado
Boulder, Colorado
wessels@colorado.edu

Abstract

The continued increase in demand for information services on the Internet is showing signs of strain. While the Internet is a highly distributed system, individual data objects most often have only a single source. Host computers and network links can easily become overloaded when a large number of users access very popular data.

Proxy-caching is currently a popular way to reduce network bandwidth, server load and to improve response time to the user. The original caching proxy, from CERN, is probably still the most widely used. This paper describes software developed by the author that investigates some alternative techniques for caching World-Wide Web objects. This software complements traditional proxy-caching by allowing servers to explicitly grant or deny permission to cache an object, and with support for server-initiated callback invalidation of changed objects.

1 Introduction

Statistics collected on the NSFNET backbone were indicating that World-Wide Web traffic would exceed that of any other protocol around May 1995. In March of this year, overall traffic is falling off as the backbone is being dismantled. Nonetheless, WWW byte counts have caught up with FTP and together they account for half of all traffic on the NSFNET.

As the World-Wide Web continues its exponential growth, many sites and network links are experiencing severe overload. Host machines at popular sites become overloaded because a large percentage of clients always request documents from the source. At one point the providers of certain dubious, but very popular materials begged for patience until their Internet provider could "widen their pipe." Another cause of load problems is less-than-optimal server software designs.

A partial solution to network and server overload is the use of network caches. Studies, simulations, and real-world experience have shown that WWW caches can significantly reduce wide area network bandwidth and improve latency [1, 2, 3, 4].

2 Background

2.1 The Protocols

The Hypertext Transfer Protocol (HTTP) protocol functions much like those which came before it (i.e. FTP, Gopher). The user runs a WWW *client*, also known as a *browser*. Popular clients are NCSA Mosaic, Netscape, and Lynx. Organizations "publish" information by running a HTTP *server*. Clients make TCP a connection to servers and request a named object. Uniform Resource Locators (URL's) are the names which are used to fully identify objects throughout the Internet. If the requested object exists it is delivered to the client and the connection is torn down.

Requests need not always be made directly between the client and server; a *proxy* can be used in between them. The client makes its request to the proxy. The proxy forwards the request to the server and then relays the resulting data back to the client.

Since a proxy is relaying requests and responses between clients and servers, it makes sense for the proxy to *cache* some of the data it has handled. A subsequent request for cached data can be given directly to the client without connecting to the server. This results in a savings of wide area network bandwidth and usually an improvement in latency to the user.

In general it is possible to cache objects which are retrieved with the HTTP, FTP, and Gopher protocols. Of these, FTP is the most difficult because the protocol is more complicated and oriented towards interactive operation.

Keeping a set of cached objects up-to-date is a difficult problem. Servers are stateless so there is no mechanism for them to push updates to clients. Instead clients must occasionally poll the servers for changed objects. When the proxy is responsible for maintaining cache consistency, it must select time-to-live (TTL) values for its objects. Worrell [2] investigated the update patterns of WWW objects and concluded that a single TTL applied to all objects works poorly.

Instead of a uniform TTL for all objects, it may be possible to set a TTL based on the object's age. Intuitively, older objects are less likely to change than younger ones. Conversely, an object which just recently changed is likely to change again soon. This approach is known as a *weakly consistent* cache. It is used by the Alex [5] FTP-to-NFS gateway, and by others. Cached objects are guaranteed to be no more than some percent stale (e.g. 10%). The *staleness* factor can be defined as

$$staleness = \frac{now - last\ update}{last\ update - last\ modification}$$

This means that an object which is 30 days old will be 10% stale after three days.

Most Internet users would probably find a staleness factor of 10% acceptable. As proxies and caches are more widely used and users become more aware of them, they may find even higher percentages acceptable. Cache consistency is less of an issue when users have the ability to force a refresh of possibly stale data. To this end, it would be nice if all WWW browsers supported "force refresh" and/or "bypass proxy" features.

HTTP enables and simplifies object caching in a number of ways: 1) Object metadata is returned with the object. 2) The metadata definition includes `Expires` and `Last-modified` fields. The first specifies an object's expiry time explicitly. The second allows an expiry time to be derived from the object's age. 3) The conditional `GET` request includes an `If-modified-since` field. The server only sends the full object data if it has been modified since the given time.

A `Last-modified` field will usually be included for objects that correspond to disk files. It is never sent with dynamically generated objects such as directory listings and CGI scripts. The `Expires` field is almost never used in practice. Of some 28,000 URL's passing through a caching proxy at Digital Equipment Corp. only six contained this field [6].

It is difficult, but not impossible, to get object metadata for FTP. File size and modification times can be extracted from directory listings, but listings are not standardized and vary between operating systems. Fortunately, more and more FTP servers support the `MDTM` and `SIZE` commands which return file size and timestamp information in a standard format [7].

The Gopher protocol does not have any support for sending object metadata.

2.2 The Tradeoffs of Caching

The three big advantages of caching WWW objects are reducing latency, network bandwidth, and server load. Another positive side-effect is the opportunity to analyze an organizations usage patterns. On the down side, a proxy introduces a single point of failure. It also requires additional resources (CPU, disk, personnel) and oversight. There is always the chance that a user will be sent stale data from the cache. A disadvantage for servers is that they do not log cache hits.

When is caching effective? A couple of conditions should be met in order for caching to be worth the expense. First, retrieving from the cache must be as fast, or faster, than from the original source. This implies that cache placement in the network is important. It should be near an organization's connection point to the Internet. Second, on average an object should be requested again while it exists in the cache. If an object changes more frequently than it is accessed, caching would be pointless. Cached objects should have long lifetimes relative to their frequency of access.

The author believes that cache hits occur for two distinct reasons. Often users re-request an object in a short period of time as they browse back-and-forth. This is an especially common practice while using the command line `ftp` client program. Directory listings are repeated as lines disappear off the top of the screen. Cache hits also occur when multiple users access the same objects.

Studies of FTP access patterns [8] have shown that 40% of transfers were of files which had been previously transferred. The same study also concluded that about 13% of all NSFNET backbone traffic was due to duplicate file transfers.

2.3 CERN Proxy Cache

The HTTP server developed by CERN[1] can also function as a proxy and as a cache. Indeed, it is the original WWW proxy and has set a number of de-facto standards for proxying and caching on the Web.

The CERN server's operation as a proxy cache is outlined in Figure 1. It has a number of parameters [9] that affect its performance and behavior. It has three methods of determining an object's expiry time. In order, they are:

1. Use the `Expires` value if present.

2. Use the `CacheLastModifiedFactor` parameter to calculate the object's TTL as a fac-

[1] `http://www.w3.org/hypertext/WWW/Daemon/Status.html`.

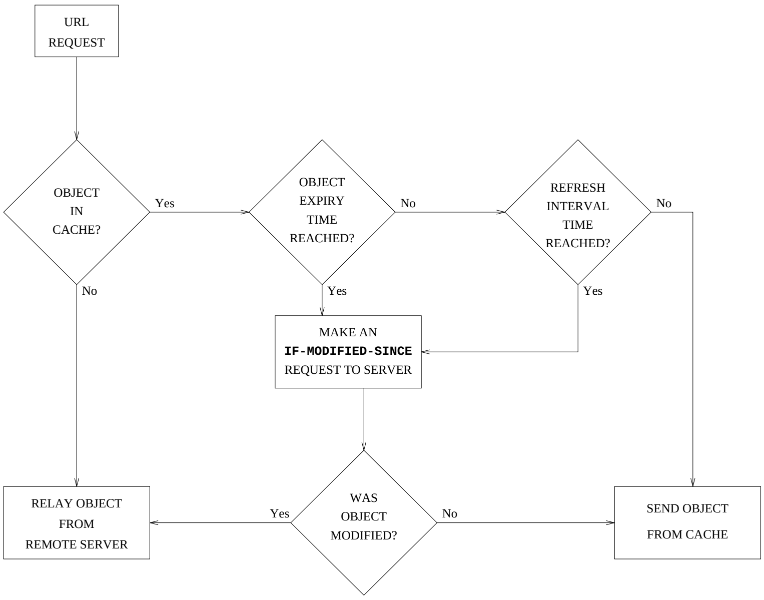

Figure 1: CERN proxy flow diagram.

tor of its age. This is essentially the weak consistency method used by Alex.

3. A `CacheDefaultExpiry` parameter may used whenever the `Last-modified` field is missing. It specifies a default, absolute time-to-live for the object. It should be zero for HTTP objects to prevent dynamic objects (scripts) from being cached.

The `CacheRefreshInterval` parameter specifies how stale an object can be until a conditional `GET` is used to make sure it is up-to-date. If the interval has not been reached, the object is sent directly from the cache. This parameter is an absolute value like "one week" and not a fraction of its age.

The `CacheTimeMargin` parameter gives the minimum age for an object to be cached. All parameters except `CacheTimeMargin` are applied to a pattern of URL's. This allows FTP objects to be treated differently from HTTP objects, for example.

3 A Different Approach

The author has developed a suite of programs to investigate a different approach to caching WWW objects. There are three components (indicated by asterisks in Figure 2):

- `web-proxy`: The proxy process. Designed to be small, fast, and efficient. It never forks and uses non-blocking I/O to service multiple simultaneous connections. Currently it supports proxying for HTTP and Gopher URL's.
- `cache-mgr`: A cache management program that runs in conjunction with the proxy. It adds, updates, and expires cached objects at regular intervals.
- `rcached`: A daemon process which can be run alongside HTTP, FTP, and Gopher servers at remote sites. Its two functions are to handle requests for permission to cache objects, and to notify cache sites of objects which have been updated or removed.

To simplify some implementation details, a two-level cache is used by these programs. The proxy copies every cacheable object into the short-term cache. Objects stay in the short-term cache until the `cache-mgr` program either moves them into the long-term cache, or simply deletes them when they expire. The cache administrator can set short-term TTL values based on regular expressions matching URL's. Objects are stored in the filesystem with a pathname derived from the URL. Because cached objects are stored on disk, they are persistent across machine failures.

Objects are placed into the long-term cache by the cache manager. Before caching an object, `cache-mgr` first checks for permission at the server site. This is done by sending a UDP query packet to the remote `rcached` process. The query includes a requested TTL value—that is, how long the proxy would like to cache the object for. If a reply is returned, it is honored. Otherwise, the object is cached anyway (as is the current convention).

The `rcached` program answers caching permission requests as they arrive. Using a simple

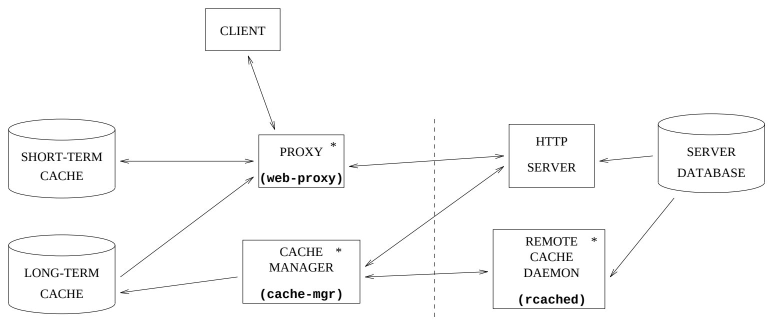

Figure 2: Interaction of Proxy/Caching components.

lookup table based on regular expressions, it responds with a "yes" or "no." When the reply is "yes," `rcached` also includes a TTL value. It may or may not be the same as was requested. This TTL value is how long `rcached` will remember that the proxy site has the object cached. Administrators may have additional insight about the nature of their data and can configure `rcached` accordingly. For example they might give relatively long TTL's to a group of read-only objects which will never change.

The cache manager will periodically refresh its cached—but unregistered—objects. An update procedure runs at regular intervals. It scans the list of cached objects and calculates each one's staleness factor. For objects whose staleness exceeds some threshold (e.g. 10%), `cache-mgr` will make a conditional `GET` request for the object.

The rest of this section discusses the reasoning and issues surrounding the design of this software.

3.1 A Simpler Proxy

All of the CERN server's functionality is bundled into a single, large executable.[2] By separating the two, the proxy process can be much smaller and simpler in design.

`web-proxy` reads objects from both the long- and short-term caches. It writes objects into the short-term cache when the following conditions are met:

- The object was requested with the `GET` method.
- The server sends a "`200 Ok`" reply.
- The object is older than some minimum age.
- Authorization was not required to retrieve the object.
- No network errors occurred during data transmission.

3.2 Reduction of System Load

The CERN server forks a new process for each proxy (and regular) request. The Unix `fork()` system call is relatively expensive. The operating system of a busy proxy site will spend too much time spawning new processes and not enough time relaying data.

Multiple connections and non-blocking I/O are handled by `web-proxy` with the use of state variables and the `select(2)` system call. Input and output functions are registered with each active file descriptor.

3.3 Optimization for Low Latency

For some client requests, the CERN proxy may issue an `If-modified-since GET` to verify an object's freshness. The connection setup and data transfer cause additional delay.

Rather than issue conditional `GET` requests, `web-proxy` assumes any object found in its cache is up-to-date. Servers are never contacted for cache hits. Maintaining cache consistency is strictly the responsibility of `cache-mgr`.

Because a proxy server handles a large number of connections, DNS queries can be a significant source of delay. Because of this, `web-proxy` uses an internal DNS cache with configurable timeouts.

3.4 Caching Permission

Caching raises some interesting issues for WWW server providers. For one, they may not appreciate the fact that *their* data is being held in caches throughout the Internet. This may be due to a misunderstanding of client/server systems. Perhaps they would believe that the data always "stays" on the server and is never stored by the client.

Some server administrators may want to en-

[2] On SunOS, 500k stripped and dynamically linked; vs. 50k for the FTP server.

sure that users are never shown stale versions of their documents. This may be important for weather-related information, stock quotes, and other time-sensitive data.

Another legitimate concern is that cache hits are not logged at the server site. People are naturally curious about how "popular" their Web pages are. Corporate sites may like to use server statistics to justify the service.

As the Internet becomes more transaction based, where the serving of Web objects means revenue, caching of paid-for objects will be totally unacceptable.

The current suite of protocols do not facilitate selectively marking some objects as uncacheable. They only real way to do this now is to somehow not send a `Last-modified` field for objects which should not be cached.[3]

A simple solution would be an addition to the HTTP protocol. Servers could include a "`Pragma: no-cache`" line in the response header.[4] A mechanism such as this could possibly be included in the next generation of the protocol: HTTP-NG [10].

Caching permission could be considered an exercise in futility because it in not enforceable. Any person or program that is able to retrieve the object data is certainly able to store a copy of it for later use. For this reason, administrators could not totally rely on this mechanism to prevent all caching of their documents.

In spite of that, `rcached` supports a simple permission mechanism that is separate from any particular server or protocol. It can be used simultaneously with HTTP, FTP, and Gopher servers on the same host.

In order for servers to log cache hits, the proxy could send off small UDP packets when a cache hit occurs. The UDP packet would contain all the information that is normally logged by a HTTP server. A modified HTTP server or `rcached` could receive and log the "hit packet." Alternatively the cache manager could support a logfile query mechanism whereby servers could request all logging data for hits to their objects. This has some serious security implications, however.

3.5 Callback Invalidation

Research has shown that server-initiated invalidation of stale objects can significantly reduce wide-area bandwidth requirements [3]. However, invalidation is generally termed "difficult" because it would be necessary to modify the server. This software proposes to *rely* on the proxy for cache consistency, but *take advantage* of invalidation when available to provide fewer stale objects. Accommodations are made for the fact that the Internet as a whole is unreliable. The goal of using invalidation here is to only improve on cache consistency. When invalidation fails, the proxy's standard consistency controls will act as a backup.

The `rcached` program acts as a registry for cached objects. When the cache manager requests permission to cache an object, it is also asking for the cached object to be registered at the server site. If the registration is accepted, `cache-mgr` will not worry about the object until it expires, or until it receives and invalidation message. Section 4 describes the registration and invalidation protocol in detail.

3.6 Tunable Cache Parameters

Proxy cache administrators should be able to tune the cache to meet their needs. Caches may not always run on hosts with a sufficiently large amount of disk space. In this case it would be desirable to to rank cacheable objects based on their frequency of access, recency of last access, and object size.

The `cache-mgr` program calculates a *cache metric* based on these three variables. The cache administrator can set values for exponents in the equation

$$m = F^f R^r S^s$$

F, R, and S are the object's frequency, recency, and size. The exponents f, r, and s determine the relative importance of each variable in the metric. f should be positive so that more frequently accessed objects have a higher precedence. Similarly, r should be negative so that more recently accessed objects (smaller R) have a higher precedence. The value of s can be positive to favor larger objects, or negative to favor smaller ones.

4 A Cache Negotiation Protocol

The cache negotiation protocol is a simple query-response exchange. There is very little data to be transmitted back and forth. The use of a reliable TCP connection has excess overhead in connection setup costs. UDP is more suited for

[3] This can be done by making the object a CGI script or by customizing the server source code. Also, the NCSA HTTP server doesn't include `Last-modified` for parsed HTML files.

[4] Note that `Pragma: no-cache` is already supported in the HTTP client request headers.

this task, except that it is unreliable. This negotiation protocol has been designed to be implemented with UDP. Therefore it must be tolerant of dropped packets and network failures. At present, approximately 10% of UDP packets are dropped when traversing the Internet.

4.1 Permission to Cache

1. The proxy $\mathcal{P}$ selects object $\mathcal{O}$ for caching. The source of $\mathcal{O}$ is server $\mathcal{S}$. $\mathcal{P}$ sends a *cache permission request* packet to the `rcached` process $\mathcal{R}$ at $\mathcal{S}$. The request packet contains $\mathcal{O}$'s URL, a checksum of $\mathcal{O}$, a requested TTL $T_{\mathcal{P}}$, and $\mathcal{P}$'s address.

   $\mathcal{P}$ waits a certain amount of time for a reply. If no reply is received, then $\mathcal{O}$ is cached as usual.

2. When $\mathcal{R}$ receives the request, it generates a *cache permission reply* packet. The reply is either a positive or negative acknowledgment of the request. $\mathcal{R}$ compares the checksum of $\mathcal{O}$ to the request and denies it if they do not match. If the reply is positive, $\mathcal{R}$ includes a TTL value $T_{\mathcal{R}}$. The TTL value suggested by $\mathcal{R}$ may be longer, shorter, or the same as the value requested by $\mathcal{P}$. In any case it is *the* TTL value as far as $\mathcal{R}$ is concerned.

   $\mathcal{R}$ logs the fact that $\mathcal{P}$ will expire $\mathcal{O}$ at time $T_{\mathcal{R}}$. It is "okay" for the reply packet to be lost in the network. $\mathcal{P}$ will assume that $\mathcal{S}$ does not run `rcached` and therefore cache $\mathcal{O}$ as usual. If $\mathcal{O}$ changes at $\mathcal{S}$, then $\mathcal{P}$ will be notified of the change—which it wasn't expecting—but will be able to deal with anyway.

3. When $\mathcal{P}$ receives a reply from $\mathcal{R}$, it marks $\mathcal{O}$ as "registered" and sets its expiry time to $T_{\mathcal{R}}$. $\mathcal{P}$ then does not need to worry about $\mathcal{O}$ until the expiry time is reached, or an invalidation message is received from $\mathcal{R}$.

4. When the expiry time is reached, both $\mathcal{P}$ and $\mathcal{R}$ should assume that $\mathcal{O}$ and information about $\mathcal{O}$ will be purged. If $\mathcal{P}$ wishes to retain $\mathcal{O}$ in its cache, this process should be restarted from step 1.

Translated into loose english, the exchange goes something like this:

$\mathcal{P}$: I've got an object $\mathcal{O}$ of yours that I'd like to keep until $T_{\mathcal{P}}$.

Possible responses are:

$\mathcal{R}$: Actually, I'd rather you didn't cache $\mathcal{O}$ at all.

$\mathcal{R}$: Sure, but while you're at it, you might as well keep it until $T_{\mathcal{R}}$. I'll let you know if it changes.

$\mathcal{R}$: Okay, but I'm pretty sure it will change before that. I'd suggest you keep it only until $T_{\mathcal{R}}$ instead.

4.2 Callback Invalidation

1. $\mathcal{R}$ periodically examines the objects which have been registered with it. When a registered object $\mathcal{O}$ has been modified, $\mathcal{R}$ sends an *invalidation* packet to $\mathcal{P}$. It is quicker to only check timestamps of registered objects, but for additional consistency `rcached` can optionally verify checksums as well.

2. When $\mathcal{P}$ receives the invalidation packet, it sends send an *invalidation acknowledgment* packet back to $\mathcal{R}$. $\mathcal{P}$ can choose to either refresh or delete $\mathcal{O}$.

3. When $\mathcal{R}$ receives the invalidation acknowledgment, it "un-registers" the entry for $\mathcal{O}$ and $\mathcal{P}$.

If either the invalidation or acknowledgment packets are lost by the network, then $\mathcal{R}$ will not un-register $\mathcal{O}$. It will send another invalidation packet during the next cycle. $\mathcal{P}$ should acknowledge this packet even if $\mathcal{O}$ has already been deleted. If $\mathcal{R}$ receives no acknowledgments after N attempts it assumes that $\mathcal{P}$ is gone and does not make any further attempts.

It is possible that registration data at $\mathcal{R}$ could be lost or overwritten. In this case $\mathcal{O}$ would remain at $\mathcal{P}$ until it expires. Because of this, server administrators should carefully consider the values of $T_{\mathcal{R}}$. A possible solution to this problem is to implement "reminder packets" which could be exchanged between $\mathcal{R}$ and $\mathcal{P}$.

5 Analysis and Results

The software has been in use at the University of Colorado for seven months. A number of users voluntarily use the cache and others use it (perhaps unknowingly) because of system wide environment variables. Since the first week of February, it has served an average of 910 requests per day. Approximately 55% of the requests are for cacheable objects. Of the requests for cacheable objects, 23.5% are cache hits. Of all requests, the cache hit percentage is only 12.8%. The cache hit rate is relatively low because for most of this time the cache disk space was limited to 15 Mb.

The software has been running on `cuboulder.colorado.edu`, a DEC AXP 3000/400 system. The `web-proxy` program imposes very little load

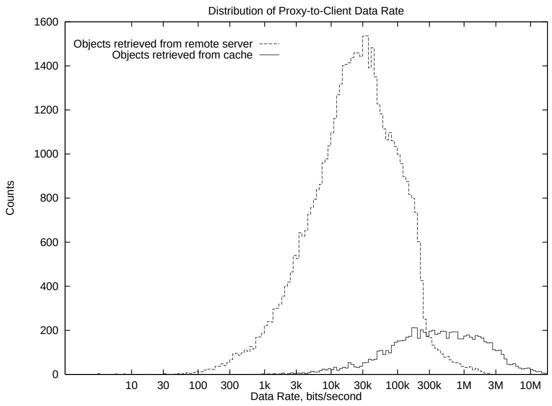

Figure 3: Distribution of proxy-to-client data rates. The mean values are 22 kbps and 360 kbps.

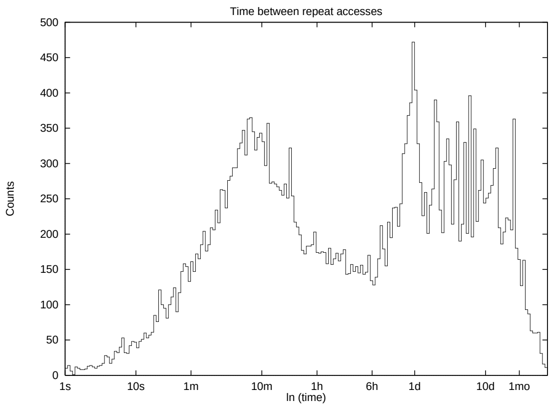

Figure 4: A histogram of the time between two accesses of the same URL. The mean is 2.75 hours.

on the host. On the average, it takes 47 milliseconds of CPU time to fulfill a client request. The `web-proxy` program compiles to 74 kbytes, and occupies approximately 400 kbytes of machine memory when running.

We can verify whether or not the proxy improves user response time by comparing proxy-to-client data rates for cached and remote objects. Figure 3 shows the distribution of this data. There is a wide variation, but in general cached objects are transferred an order of magnitude faster (22 kbps vs. 360 kbps). The graph also indicates the relative amount of objects sent from the cache.

Figure 4 shows the distribution of repeat access interarrival times. The two-hump shape of the data supports the assumption that repeat accesses occur for two different reasons. Most of the first hump is likely due to repeat accesses by a single user, while the second hump is from otherwise unrelated users accessing the same set of objects. The mean of this distribution is 2.75 hours. That the first hump peaks out around ten minutes suggests that the short-term cache alone receives a significant number of cache hits.

With the long-term cache disk space limit currently set at 20 Mb, it contains 1000–1100 objects. On `cuboulder`, where the system load average hovers around two, it takes around 30 minutes for `cache-mgr` to execute its long-term maintenance routine. This involves:

1. Removing expired objects.

2. Parsing the logfile from `web-proxy`.

3. Calculating and sorting the cache metrics.

4. Adding new objects to the cache, either by moving them from the short-term cache, or by transferring them across the network.

While it takes a half-hour of real time, it uses only one minute of CPU time. This implies that `cache-mgr` spends a large fraction of its time waiting. It blocks on DNS queries and when reading data from the network. The default timeout for making TCP connections and reading from sockets is ten seconds.

The `cache-mgr` update routine only refreshes stale objects. On `cuboulder` it runs every three hours. It takes six real minutes and 17 cpu-seconds to examine 1100 objects and refresh the stale ones. On average, 6.5% of the objects in the long-term cache are at least 10% stale. Of the stale objects, 72% are verified as not modified. The remaining 28% are either refreshed or cannot be retrieved. This implies that on average, no more than 2% of the cached objects are ever out-of-date.

At the time of this writing, the `rcached` program is being rewritten for use UDP with packets. The initial implementation used reliable TCP connections with a somewhat simpler negotiation protocol. The author feels that UDP fits better with the loose cache consistency models.

6 Related Work

6.1 Harvest

The Harvest software [11] is a scalable and customizable discovery and access system for Internet information.[5] One of the Harvest components is a hierarchical object cache [12]. The Harvest cache is extremely fast at returning cached objects because it makes very efficient use of machine virtual memory. Metadata for all objects are stored in VM at a cost of 750 bytes per object. The total amount of VM that it uses is a configurable parameter. If object metadata doesn't

[5] `http://harvest.cs.colorado.edu/`

consume all of the allowed memory, the remainder is filled with the most active objects. Measurements indicate that the Harvest cache is two orders of magnitude faster than the CERN proxy with ten concurrent clients. They attribute 80% of the improvements to the use of virtual memory.

The Harvest cache never forks for HTTP and Gopher retrievals. Because FTP is somewhat complicated, an external program is used to fetch FTP objects. Non-blocking I/O is used for all disk and network reads and writes. To avoid another potential source of blocking, it also includes a customized DNS server and cache.

Harvest caches are manually configured in a hierarchy with parents and neighbors. A cache miss results in a series of UDP query packets being sent to neighbor and parent caches. A "cache hit" packet is also sent to the UDP echo port of the object's home site. The object is retrieved from first site to return a "hit" packet, or from the first parent to return a "miss" if all caches miss. In the unlikely event that a parent cache is down, and the home site has UDP echo disabled, there is a two second timeout until the object is retrieved from the source.

The author recently became a Professional Research Assistant on the Harvest project. Some of the results from this research will be incorporated into future versions of the Harvest cache.

6.2 Push-Caching

Gwertzman [3] analyzes a technique termed *geographical push-caching*. In this model, servers use their knowledge of access patterns on their data to optimally distribute their popular objects among other geographically distributed push-cache servers. Data is distributed geographically because network hop count is reasonably correlated to geographic distance, and even more so when limited to a single backbone network.

His simulations led him to conclude that push-caching is most effective for popular servers which have a small set of "hot" documents. The server cluster at NCSA certainly fits this requirement. He also concluded that push-caching and client-caching have the same bandwidth requirements, but affect a different set of documents. Bandwidth was significantly reduced when both client- and push-caching were used together.

Gwertzman also evaluated three different methods for maintaining cache consistency. He compared a fixed time-to-live, the Alex update threshold algorithm, and server-initiated invalidation. Simulations showed that invalidation and update threshold require roughly the same bandwidth, but invalidation ideally provides no stale objects.

7 Conclusions and Future Work

The Internet will be unable to scale with current growth trends for World-Wide Web and related services. Proxy-caches will become a necessity at organizational and regional networks. If caches are not deployed for the noble goals of reducing wide area traffic and server load, then a metered scheme for charging would provide a "bottom-line" financial reason.

The traditional model of one-process-per-connection is not realistic for current and future demands on information systems. Both this and the Harvest cache software [12] demonstrate that a non-forking, multi-threaded proxy implementation can reduce server load while at the same time improving response time by an order of magnitude or more.

It is unclear how actually useful it is to request permission before caching an object. In the absence of more stringent encryption or other security mechanisms, it is somewhat pointless to deny caching permission when objects can be freely retrieved anyway.

The server-initated callback invalidation needs more testing and development. It will likely be difficult to achieve widespread use of `rcached` or similar software unless the functionality were bundled with HTTP servers, for example.

7.1 Software Availability

The software described here, and other information related to caching, is available on the Web at `http://itp-www.colorado.edu/~wessels/Proxy/`.

Acknowledgments

The author would like to thank Dr's. Jon R. Sauer, Kenneth J. Klingenstein, and Oliver S. McBryan for their continued support and assistance. Thanks to Mr. Al Marti for the opportunity to evaluate the software on the University of Colorado's computer systems. Finally many thanks to Michael Schwartz, Peter Danzig, and the other members of the Harvest development team.

This work was supported in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. The information contained in this thesis does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

References

[1] P. B. Danzig, R. S. Hall, and M. F. Schwartz, "A case for caching file objects inside internetworks," in *Proceedings of the SIGCOMM '93*, (San Francisco, California), pp. 239–248, September 1993. `ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/FTP.Caching-PS/Paper.ps.Z`.

[2] K. J. Worrell, "Invalidation in large scale network object caches," Master's thesis, University of Colorado, Boulder, Colorado, 1994. `ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/WorrellThesis.ps.Z`.

[3] J. Gwertzman, "Autonomous replication in wide-area internetworks." Senior Thesis, Harvard College, Cambridge, Massachusetts, 1995. `http://das-www.harvard.edu/research/vino/push.cache/`.

[4] K. C. Claffy, "Web traffic characterization: an assessment of the impact of caching documents from NCSA's web server," in *Proceedings of the Second International WWW Conference*, 1994. `http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/claffy/main.html`.

[5] V. Cate, "Alex—a global filesystem," in *Proceedings of the Usenix File Systems Workshop*, pp. 1–11, May 1992. `ftp://alex.sp.cs.cmu.edu/doc/intro.ps`.

[6] S. Glassman, "A caching relay for the world wide web," in *Proceedings of the First International WWW Conference*, pp. 69–76, May 1994. `http://www.research.digital.com/SRC/personal/Steve_Glassman/CachingTheWeb.html`.

[7] J. Postel and J. Reynolds, "File transfer protocol," *Network Working Group RFC 959*, October 1985. `ftp://ftp.internic.net/rfc/rfc959.txt`.

[8] D. J. Ewing, R. S. Hall, and M. F. Schwartz, "A measurement study of internet file transfer traffic," Tech. Rep. CU-CS-571-92, University of Colorado, Dept. of Computer Science, Boulder, Colorado, January 1992. `ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/FTP.Meas.ps.Z`.

[9] A. Luotonen, "Proxy caching." `http://www.w3.org/hypertext/WWW/Daemon/User/Config/Caching.html`, 1994.

[10] S. E. Spero, "Progress on http-ng." `http://www.w3.org/hypertext/WWW/Protocols/HTTP-NG/http-ng-status.html`, 1994.

[11] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, M. F. Schwartz, and D. P. Wessels, "Harvest: A scalable, customizable discovery and access system," Tech. Rep. CU-CS-732-95, University of Colorado, Dept. of Computer Science, Boulder, Colorado, March 1995. `ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/Harvest.Jour.ps.Z`.

[12] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache," Tech. Rep. 95-611, University of Southern California, Dept. of Computer Science, Los Angeles, California, March 1995. `ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/HarvestCache.ps.Z`.

[13] M. A. Blaze, *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993. `http://www.cs.princeton.edu/TR/PRINCETONCS:TR-397-92?abstract=`.

[14] D. Wessels, "Intelligent caching for world-wide web objects," Master's thesis, University of Colorado, Boulder, Colorado, January 1995. `http://itp-www.colorado.edu/~wessels/Proxy/wessels-thesis.ps`.

Author Information

The author is a Professional Research Assistant with the University of Colorado. He has a B.S. in Physics from Washington State University, and recently received an M.S. in Telecommunications at the University of Colorado. He can be reached via email at `wessels@colorado.edu` or at *University of Colorado, Campus Box 430, Department of Computer Science, Boulder, CO 80309-0430*. His home page is `http://harvest.cs.colorado.edu/~wessels/`.