DECEMBER 1999

WRL
Research Report 99/3

Errors in timestamp-based HTTP header values

Jeffrey C. Mogul

Errors in timestamp-based HTTP header values

Jeffrey C. Mogul

Compaq Computer Corporation Western Research Laboratory
mogul@pa.dec.com

December, 1999

Abstract

Many of the caching mechanism in HTTP, especially in HTTP/1.0, depend on header fields that carry absolute timestamp values. Errors in these values could lead to undetected cache incoherence, or to excessive cache misses. Using an extensive proxy trace, we looked for HTTP responses exhibiting several different categories of timestamp-related errors. A significant fraction of these responses have detectable errors in timestamp-based header fields.

Table of Contents

List of Figures

1. Introduction

The HTTP protocol allows proxies and clients to cache certain responses. Caching improves performance by reducing latency and bandwidth consumption, and may increase availability. However, caching can lead to errors if a cache returns a response that is not consistent with the response that would have been returned had no cache been present. (This is referred to as a failure of the cache's ‘‘semantic transparency.’’) For this reason, the HTTP protocol includes several features intended to preserve the semantic transparency of caches.

Several of these features, especially in HTTP/1.0, depend on message header fields that carry absolute timestamp values. The use of absolute timestamps runs a risk if server, client, or proxy clocks are set incorrectly. Errors in these timestamp values could lead to undetected cache incoherence, or to excessive cache misses. The success of certain aspects of HTTP caching depends on the accuracy of these clocks, as did some of the protocol design decisions for HTTP/1.1. However, relatively little information has been published about the prevalence of clock errors in actual Web usage.

We have obtained some information about clock errors in HTTP messages by studying an extensive proxy trace. We looked for HTTP responses exhibiting several different categories of timestamp-related errors, some of which could lead to transparency failures; others might lead to less use of caching than necessary. We also identified several cases in which the problem may not lie with an incorrect clock per se, but with the server software that generates the response header fields.

The cache-related timestamps in HTTP suffer from an additional problem that can result in transparency failures. These timestamps, and in particular in the Last-Modified response header, have a resolution of one second. Thus, if the content associated with a URL changes twice during one second, the Last-Modified mechanism cannot always detect this. This lack of resolution can lead to transparency failures even when all clocks are precisely synchronized. Because our traces include cryptographic digests of the HTTP response bodies, we can detect such failures, and therefore quantify their prevalence.

We will show that a significant fraction of the HTTP responses in our trace do indeed have detectable errors in timestamp-based header fields. For example, 38% of the Date headers show times in the future, relative to the times at which they were logged on a system with an accurate clock.

2. Prior and related work

Several previous studies have reported on various aspects of HTTP protocol correctness, and on clock synchronization.

Krishnamurthy and Arlitt [3] studied what fraction of popular Web sites implement HTTP/1.1, and of these, how well these servers appear to comply with RFC2616, the HTTP/1.1 specification [2]. They did not address any issues related to timestamp correctness or to correct caching behavior.

The Web Replication and Caching Working Group (WREC) of the Internet Engineering Task Force (IETF) has cataloged a number of known problems with Web proxy caches [1]. The current version does not list any problems related to timestamps in HTTP messages, although it does point out that insufficient timestamp resolution in server and proxy logs can interfere with log analysis.

Several studies have reported on time synchronization in the Internet, using the facilities provided by the Network Time Protocol (NTP) [4]. The largest and most recent of these surveys, by Minar [5], reports ‘‘a surprising number of bad timekeepers.’’ That is, many ostensibly synchronized clocks are not. This study did not look specifically at Web server clocks.

Wills and Mikhailov [7] studied a set of URLs taken from a proxy cache log, fetching multiple versions of these URLs over a period of days. This allowed them to compare the Last-Modified timestamps on the responses, and MD5 digests of the response bodies, to detect when the Last-Modified timestamps would give either non-transparent results, or would cause unnecessary cache misses. They found that a small fraction of potential transparency failures, as well as a much larger fraction of unnecessary cache misses.

3. Trace collection

We obtained our traces at the Palo Alto, California proxy of Compaq Computer Corporation, one of several firewall proxies serving the company. The proxy is used for access control, not for performance, and so is not set up as a cache.

The proxy runs version 1.1.20 of the Squid proxy software [6]. We modified Squid to compute an MD5 digest for the body of each response, and to log these digest values in one of the log files Squid already keeps (`squid.store.log`). We also augmented this log format to include the connection duration, in milliseconds, as measured by the proxy. These changes were relatively simple, but Squid is a complex program and there may be a few error conditions in which we could log the wrong digest value.

Each `squid.store.log` entry also includes a timestamp (with roughly millisecond resolution), status information, the length of the response body, and the values of selected HTTP response header fields: ‘‘Date’’, ‘‘Last-Modified’’, ‘‘Expires’’, and ‘‘Content-Type’’. Unfortunately, it does not include the value, if any, of the ‘‘If-Modified-Since’’ request header; this makes it impossible to completely model the behavior of a caching proxy.

The log entries contain complete URLs with server host names (such as `http://www.compaq.com/`), rather than server IP addresses. A given name might resolve to several IP addresses, allowing transparent server replication; we assume that any site with this feature is internally consistent.

The average log entry consumes about 169 bytes, or about 44 bytes after compression with gzip. It is thus feasible to store many millions of log entries on a moderately large disk.

We collected a continuous trace covering covering 90 days from 1 January 1999 through 31 March 1999. This trace includes 125,259,641 log entries, and occupies 5.1 GBytes in compressed form. The busiest day during the trace accounts for 2,085,909 log entries.

4. Trace analysis

We wrote a program to analyze a trace. It starts by parsing the extended `squid.store.log` format, skipping over unusable log entries. These include malformed entries, aborted transfers, all HTTP methods other than ‘‘GET’’, and all HTTP response status codes other than 200 (the normal success code, where the response carries the entire body of the resource value).

The program also skips a relatively small set of responses which match both the URL and MD5 digest of a previous response, but which have different content-type or content-length values (according to the log entries). Since it should not be possible to generate the same MD5 digest if the length varies, we believe that these ‘‘impossible’’ values represent failed transfers that are not indicated as such in the log. A review of the Squid sources reveals several possible places where a transfer could be aborted without being logged as such (this is our fault, not a bug in Squid). Some of the content-type mismatches may reflect a changed content-type assignment at the server, perhaps because of a misconfiguration, but we also try to skip these to avoid confusing the results.

From the 125,259,641 log log entries in our trace, this winnowing process yielded 79,441,708 entries (about 63%) usable for our analysis.

The entire trace included useful responses from 459,225 distinct server host names. Our logs include approximately 25,000 unique clients.

The analysis program uses each entry to create one or more nodes in an *ad hoc* database. For example, a record for each unique MD5 digest D_i is stored in a hash table. This record serves as the head of a list of nodes each representing a tuple (D_i, URL_j), where URL_j is a distinct URL with at least one response matching digest D_i. Each such node contains back-pointers to a nodes describing URL_j in more detail, and itself serves as the head of a list of nodes with per-entry information (such as elapsed time).

Once the database has been created, the linkages between the various nodes allow the analysis program to traverse the database in various orders, collecting statistical information about the responses described in the trace.

4.1. Cache simulation

Although we obtained our trace at a non-caching proxy, our analysis program attempts to simulate the behavior of a caching proxy, which allows us to detect cases where a real cache might suffer from transparency failure. However, we use an atypical approach to HTTP cache simulation.

A typical HTTP cache uses two mechanisms to increase the likelihood that it will provide accurate responses:

1. **Expiration times**: If the server provides an ‘‘Expires’’ header, the cache assumes that the response can be used until that deadline, without refreshing it from the server. Otherwise, some caches estimate an expiration time, based on the ‘‘Last-Modified’’ time and other parameters.

2. **Revalidation**: If a cache entry has expired, the cache checks with the server to see if the entry is still valid. It does this check by sending a GET request with an ‘‘If-Modified-Since’’ header including the ‘‘Last-Modified’’ date from the cached response; if the resource is unmodified, the server responds with a status code of 304 (‘‘Not modified’’).

At a proxy cache, which acts as both client and server, the picture is more complicated. For example, a proxy that receives a GET request with an ‘‘If-Modified-Since’’ header might find in its cache a matching and unexpired entry (and should return a 304 response), or a more recent unexpired entry (and should return a 200 response), or it might have to forward the request towards the ‘‘origin server’’ (the master server for the resource).

The HTTP caching mechanisms do not guarantee that caches provide accurate (that is, cache-coherent) responses, for several reasons: The expiration time might be too optimistic, or the Last-Modified timestamp might be wrong (perhaps due to clock skew), or the source might be modified twice during one second (a condition not detectable with the one-second resolution of the ‘‘Last-Modified’’ header). On the other hand, the use of timestamps to check validity eliminates certain possibilities for cache hits, especially for dynamic resources. HTTP/1.1 introduces a new ‘‘entity tag’’ mechanism to avoid some of these problems [2], but Squid 1.1.20 does not understand entity tags, and so these do not appear in our logs.

Instead of trying to emulate an HTTP cache (partly because our traces lack ‘‘If-Modified-Since’’ header values), we simulate a ‘‘perfect coherency’’ cache, where a request seen for a cached URL will always miss if the origin server would return a different value, and will always hit if the origin server would generate the exact same value. We use the MD5 digest values from our logs, and the fact that our logs (unlike those of a true cache) contain an origin-server response for every request. This allows us to test, on every request, whether the origin server’s response is identical to what would have been cached (we assume no MD5 collisions).

While a perfect coherency cache does not generate exactly the same set of hits as a normal HTTP cache, we believe that it generally will see a slightly higher hit rate (because of the potential for hits after an entry has expired, and because of the potential for hits on dynamic resources). However, this effect is largely irrelevant for the purposes of this study, since we do not actually report cache hit rates.

If one assumes an infinite cache, the perfect coherency simulation is also trivial to implement in our analysis program: if any two responses for URL_j have the same digest value D_i, then the second is a cache hit. If a response for URL_j arrives with a unique digest value, then it represents a cache miss. We decided not to simulate finite caches for other reasons (chiefly, the need to choose and implement a realistic replacement policy), so we avoid the complexity of simulating a finite perfect-coherency cache. However, in our simulation a cache entry is deleted if a newer entry is created for the given URL.

5. Results of error analyses

We applied several analyses to our traces, each of which looks for anomalies in the timestamp-related HTTP response headers.

5.1. Date skew

Every HTTP response should include a Date header, showing the time at which the response was originally generated. HTTP/1.1 makes this an explicit requirement, but some older servers do not always send a Date header. In our trace, approximately 89% of the useful trace entries included a parsable Date header.

Because our proxy server uses NTP to synchronize its own clock to an absolute reference (with an accuracy of several milliseconds), we can compare the Date headers recorded in the trace with the corresponding trace timestamps, to see if the servers that generated these Date headers had properly synchronized clocks.

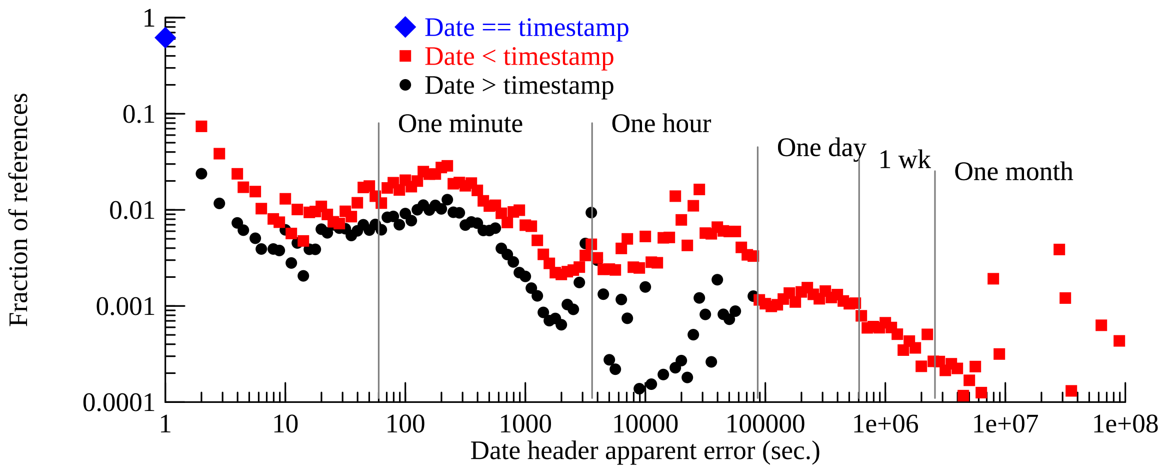

Figure 5-1: Apparent errors in HTTP Date response headers

Figure 5-1 shows two distributions on one graph. The horizontal axis shows the absolute value of the apparent ‘‘error,’’ calculated as the difference between the received Date header and the trace entry timestamp. The mark on the vertical axis shows that about 62% of the Date values were within about 1 second (the best-case measurement accuracy) of the correct time.

The square marks show the distribution for Date values that were older than the timestamp; these are not necessarily erroneous, since the response in question could have been stored for some time in an intermediate cache. (Some of the short-term errors could also be due to response transfer time.) However, we believe that some of these ‘‘Date < timestamp’’ values are indeed erroneous, although this kind of error will not cause HTTP caches to return stale values.

The circular marks, however, represent real errors: Date values that appear to be in the future. One can ascribe some of the short-term errors to minor clock mis-synchronization, but a large fraction (about 1%) are exactly one hour fast, and a modest fraction (about 0.1%) are about a day fast. A very small set of Date headers (below the *y*-axis threshold for this log-log graph) have even larger errors, ranging up to years or even decades.

It may be hard to tell from the distribution in figure 5-1 just what fraction of HTTP Date header values are seriously in error. Figure 5-2 shows the cumulative distribution of the error for impossible Date values; that is, Date values that are still in the future when received by the tracing system.

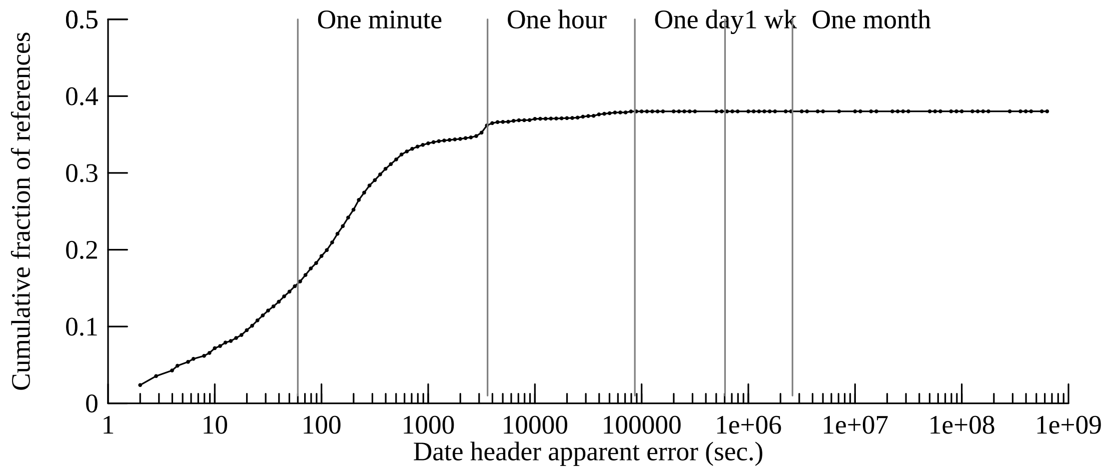

Figure 5-2: Apparent future-date errors in HTTP Date response headers (cumulative)

A total of 38% of the Date values are at least one second in the future. More significantly, 22% are at least 60 seconds in the future, and 1.5% are at least an hour in the future. However, only 0.01% are a day or more in the future. It appears that while many server sites set their clocks sloppily, and some sites seem to be confused about their time zone (or whether daylight savings is in effect), most sites do make some effort to set a plausible clock time.

5.2. Impossible Last-Modified values

If an HTTP origin server is willing to let a response be stored in a cache, it may supply a Last-Modified response header giving the timestamp at which the resource was most recently modified. A cache can later validate a stored response by asking the server (using the If-Modified-Since request header) whether the resource has been changed since the Last-Modified value for the cache entry. Most recent implementations require strict timestamp equality in order to consider a cache entry as valid, but some early implementors tried to infer additional information from the Last-Modified value.

One would not expect the Last-Modified value in a response to be any newer than the response's Date value. In practice, many server implementations obtain these values from different clocks: for example, an HTTP server that mounts a remote file system via NFS would obtain a file's modification timestamp from the file server's clock, not from its own. As long as the file-modification timestamp is consistent, this should cause no problem for strict-timestamp-equality cache validation, but it might cause problems for other uses.

71% of the useful references in the trace had parsable Last-Modified headers. Figure 5-3 shows that all but about 0.3% of these were reasonable, with most of the remaining dates off either by about 4 minutes, or by somewhat less than a year. It may be that a small set of servers accounts for these two spikes in the distribution. In any case, unreasonable Last-Modified dates are rare, but they are seen in practice.

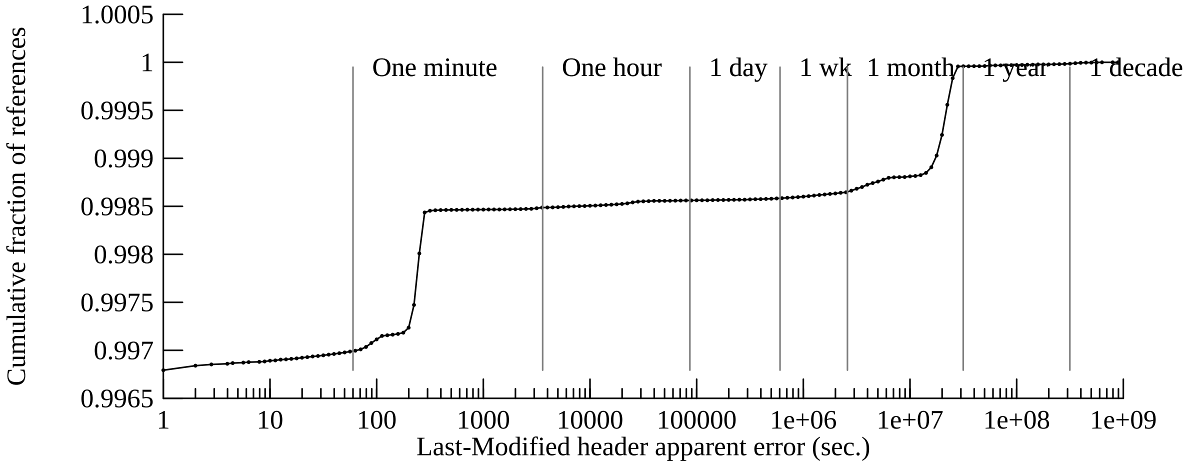

Figure 5-3: Apparent errors in HTTP Last-Modified response headers

5.3. Apparent coherency failures using Last-Modified

Implicit in the use of the Last-Modified and Get-Modified-Since headers is the expectation that if the Last-Modified value for a URL has not changed, then the value of the URL has itself not changed. The presence of a Last-Modified header is also an implicit signal that the origin server is willing to have the response cached, since otherwise the header has little practical significance.

Because our traces include both Last-Modified values and MD5 digests, when the trace includes two or more responses for a URL, we can check each but the first response to see if it represents a coherency failure. That is, we can check for cases where the Last-Modified values match, but the response bodies themselves do not.

Note that such coherency failures do not necessarily represent implementation errors; some such errors will occur because of event sequences that cannot be safely handled by the HTTP/1.0 protocol. For example, if a resource changes twice or more during one second, two different instances may have identical Last-Modified timestamps.

Among all of the responses in the trace carrying a Last-Modified header (71% of the useful references), 36% had no predecessor with the same Last-Modified value. Mostly there was either no predecessor at all, or a predecessor with a different value, but in a few cases the previous response might have omitted the Last-Modified header.

61% of the responses with Last-Modified headers were coherent; that is, they had both the same Last-Modified value and the same MD5 digest as a prior response for the same URL. This might not have been the immediately prior response, however.

About 3% of the responses with Last-Modified headers were incoherent; that is, they had the same Last-Modified value as a prior response for the same URL, but not the same MD5 digest. This seems like a surprisingly high rate of coherency failure, but it is possible that in many of these cases, the responses are semantically equivalent. For example, they might be equivalent except for a randomly chosen advertising banner.

HTTP/1.1 [2] makes a distinction between ‘‘weak’’ and ‘‘strong’’ cache validators. For a strong validator, the protocol absolutely requires that identical validators imply identical response values. The evidence from our traces suggest that Last-Modified dates should definitely be treated as weak validators.

Wills and Mikhailov [7], in their similar analysis, used a smaller trace, which would therefore be more susceptible to cold-cache effects. They report results only for two subsets of their trace, one including resources referenced at least 100 times, and the other including resources referenced between 20 and 99 times. They found only 0.05% of the references in the former subset, and 0.73% of those in the latter subset, exhibit ‘‘weak’’ Last-Modified times (i.e., the Last-Modified value remains the same, but the MD5 digest changes). It might be possible to infer, from their results and ours, that the weakness of the Last-Modified validator decreases with increasing popularity.

5.3.1. Effects of limited timestamp resolution

The Last-Modified header has a resolution of 1 second. Therefore, even a correctly implemented server can generate equal Last-Modified values for different content if a file is updated more than once during a second. In a pair of responses that exhibit this problem, the Date header value and the Last-Modified value would have to be identical for at least the first response. This is because the first Date header should be generated after the first Last-Modified time, and before the second Last-Modified time, and the two Last-Modified times are equal in this case.

In our trace, errors caused by the limited resolution of the Last-Modified header seem to be quite rare. Among all pairs of responses, for a given URL, where both carried Last-Modified errors, only 0.01% appear to have suffered from a coherency error (same Last-Modified value, different MD5 digest) that could have been caused by the limited resolution. Only 0.2% of the coherency errors could be ascribed to this effect. While one might expect a slightly higher ratio in a trace representing many more clients, and thus more frequent references to individual URLs, it appears that the limited timestamp resolution is an insignificant contributor to coherency errors.

5.4. Apparently premature Expirations

An origin server may assign an expiration time (via the Expires header, or the HTTP/1.1 ‘‘Cache-Control: max-age’’ directive), after which time a cache is expected to revalidate a stored response before using it. Servers usually assign expiration times heuristically, guessing how long it might be before the resource is changed.

A server has an incentive to assign a conservative expiration time. This is in no way a protocol error, since the alternative would be to allow inappropriate caching, but it may be interesting to determine how often this conservative policy causes cache misses. Because our traces include both Expires values and MD5 digests, we can sometimes determine whether a cached response expires before the response value changes.

Unfortunately, we can only make this check when receiving a subsequent response for the same URL (i.e., when processing a subsequent request). This means that we cannot detect all of the cases where the expiration time is too conservative, but only those where a reference in our trace would ‘‘sample’’ this status.

We start by measuring the distribution of the effective maximum age for the responses in our trace. In HTTP/1.1, the ‘‘Cache-control: max-age’’ directive explicitly specifies the maximum allowable age of a cached value (at which point it becomes stale); in HTTP/1.0, as implemented by Squid 1.1.20, an effective max-age value can be calculated as the difference of the Expires and Date headers, for those responses that include both. (This ignores the possibility of a broken server clock.)

13.9% of the useful response included an Expires header, but only 7% of the responses carried both Expires and Date headers. That is, the HTTP/1.1 rule requiring servers to send Date seems not to be universally observed. A detailed analysis of the responses carrying an Expires header but no Date header reveals that a small number of very popular servers account for most of these anomalous headers. These servers might be running specialized software.

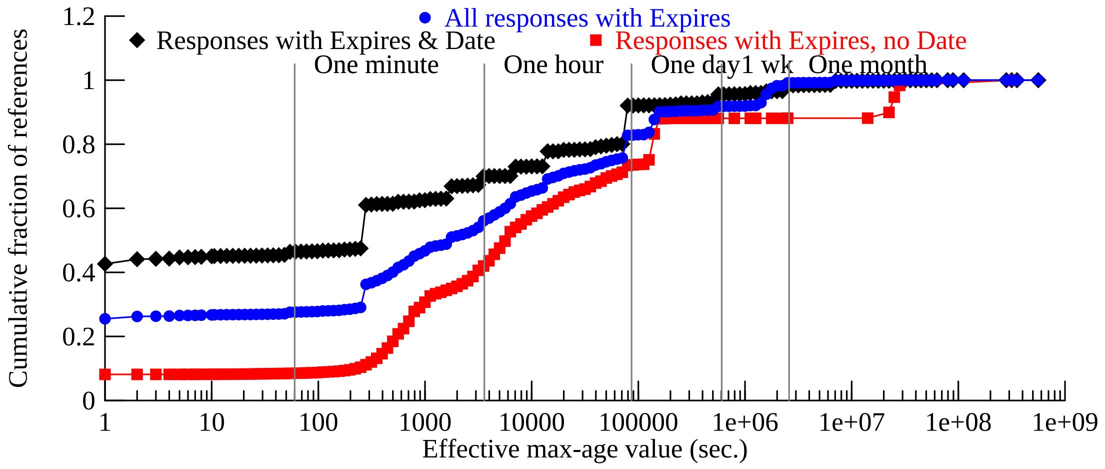

Figure 5-4: Effective max-age values for responses

Figure 5-4 shows the cumulative distributions for the effective max-age values, computed in three ways: for responses with both Expires and Date, the difference between those headers; for responses with only Expires, the difference between the Expires value and the log-entry timestamp; and the union of the two sets (which have no intersection).

Of the set of responses with both Expires and Date headers, 41% (2.8% of all responses) were ‘‘pre-expired.’’ (A pre-expired response may be stored in a cache, but the cache needs to revalidate the response before using it for a subsequent request.) This includes 17% of that set where the Expires and Date headers were set equal (the most straightforward way to mark a response as pre-expired). It is somewhat surprising that such a large fraction of the responses carrying Expires headers do so to inhibit caching, but this may be an effective way of ensuring that HTTP/1.0 caches do not violate HTTP/1.1 caching rules.

While many responses are pre-expired, how many of the remainder expire too soon? Whenever we see two responses in the trace for a specific URL with the same MD5 digest value, we can infer that the resource has not changed in the interim. (This assumption is questionable, because the resource could have temporarily changed to a different value, but we ignore this bug in our analysis.) If the expiration time for the previous response has passed, then we can call this a prematurely expired response.

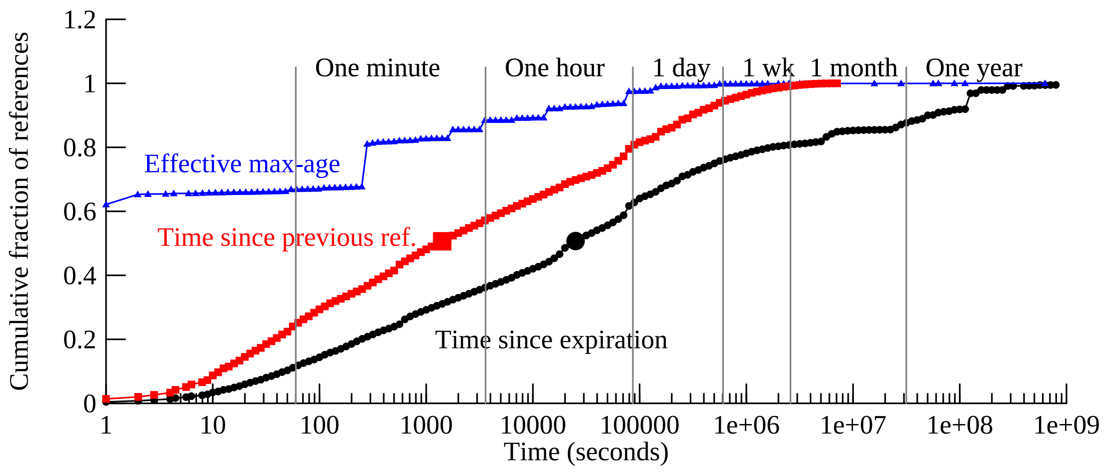

Figure 5-5: Distributions for prematurely-expired responses

Figure 5-5 shows the cumulative distributions of several values for pre-expired responses. The distribution marked ‘‘effective max-age’’ shows the calculated max-age values for pre-expired cached responses. (This includes only those responses with positive max-ages; 60.3% of these responses had negative or zero max-ages.) The distribution marked ‘‘time since previous ref.’’ shows the inter-arrival times for requests to identical URLs, in cases where the previous response has expired prematurely. The median of this distribution is about 1400 seconds (roughly half an hour).

The distribution marked ‘‘time since expiration’’ shows the amount of time that has passed since the previous response expired, measured at the moment when a new response is received with the same MD5 digest value. The median of this distribution is over 25,000 seconds (almost seven hours), much larger than the median of the interarrival times; this is because many pre-expired responses have Expires values much older than their Date values.

Of course, premature expirations do not lead to coherency failures, and it may be necessary to underestimate the expiration time of a response in order to prevent coherency failures from unexpected modifications. Premature expirations do, however, reduce the efficacy of HTTP caches.

5.5. Over-optimistic Expirations

While setting an excessively conservative expiration time does not result in a transparency failure, setting an excessively generous expiration can cause incorrect caching. We can detect some instances of this error: when we see two responses in the trace for the same URL, but with different MD5 digests, if the expiration time of the former response has not passed when the latter response arrives, then that expiration time is too optimistic.

Out of all of the usable entries in our trace, 5,041,078 (6.3%) were for a URL where we had seen a previous response with a different MD5 digest. That is, in these cases, an unexpired cached response would be incoherent.

Of these potentially incoherent cache entries, 2,528,560 (3.2% of all entries) had already expired by the time that the newer response arrived. However, 2,512,518 (also 3.2% of the complete set) had not yet expired, and so could have been used in error. (Typically, any response carrying an Expires header is presumed to be cachable.)

It may seem remarkable that 3.2% of the requests in our trace would have received ‘‘wrong’’ responses, had a sufficiently large proxy cache been in use. However, it appears that many of these apparent incoherencies are harmless. For example, a popular and cachable HTML page may change, on every reference, the embedded URL for an advertising banner. A client of a caching proxy sees the right content, but perhaps not the right advertisement.

Unfortunately, because our trace logs contain only the MD5 digest values, and not the individual HTML files themselves, it is not possible to verify that this kind of harmless incoherency is, in fact, the main cause the high rate of ‘‘wrong’’ responses. In some cases, there might indeed be true incoherencies.

Note that the rate of responses with over-optimistic expiration dates is about the same as the rate of responses whose Last-Modified headers are incoherent (see section 5.3). These sets may have considerable overlap; for example, a server that sets a lengthy expiration period for an HTML response while continually changing the embedded advertising image might not update the Last-Modified value for each ad banner change.

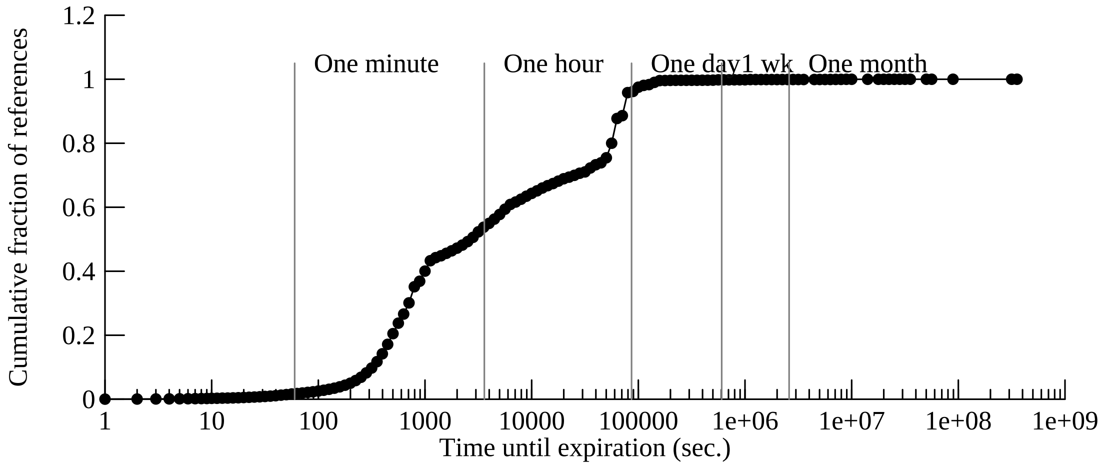

Figure 5-6: Remaining time until expiration for changed responses

Figure 5-6 shows the distribution of the time remaining until expiration for potentially incoherent cached responses, measured at the time of the subsequent request. (These are the responses that carry Expires headers and that differ from what the origin server actually returned at the time of the subsequent requests.) The distribution is generally similar to the effective max-age distribution shown in figure 5-4. However, the max-age distribution has some density at values above a few days, while figure 5-6 does not. This implies that sites which assign over-optimistic expiration times (whether by accident or on purpose) are making some attempt to limit the duration of potential incoherency.

6. Summary and Conclusions

We found that 38% of HTTP responses in our traces have impossible HTTP Date header values, and that 0.3% had impossible Last-Modified values, either of which errors could lead to transparency failures. We also found that the Last-Modified header gives an incorrect assurance of coherency in about 3% of the responses, and that the Expires header is optimistic in roughly 3% of the responses. Not all of these errors lead to transparency failures, and not all transparency failures are significant. However, the prevalence of obviously incorrect clock settings should encourage HTTP implementors to avoid using protocol mechanism based on absolute timestamps, in favor of newer HTTP/1.1 mechanisms that tolerate clock skew.

Server operators (and, probably, proxy cache operators) need to do a better job of setting their clocks.

References

[1] John Dilley. *Known HTTP Proxy/Caching Problems*. Internet-Draft draft-ietf-wrec-known-prob-00.txt, IETF, September, 1999. This is a work in progress.

[2] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, HTTP Working Group, June, 1999.

[3] Balachander Krishnamurthy and Martin Arlitt. *PRO-COW: Protocol Compliance on the Web*. Technical Report 990803-05-TM, AT&T Labs -- Research, August, 1999. http://www.research.att.com/~bala/papers/procow-1.ps.gz.

[4] David Mills. *Network Time Protocol (v3)*. RFC RFC 1305, Internet Engineering Task Force, April, 1992.

[5] Nelson Minar. A Survey of the NTP Network. December, 1999 http://www.media.mit.edu/~nelson/research/ntp-survey99/.

[6] Duane Wessels. Squid Internet Object Cache. http://squid.nlanr.net/Squid/.

[7] Craig E. Wills and Mikhail Mikhailov. Examining the Cacheability of User-Requested Web Resources. In *Proc. 4th International Web Caching Workshop*. San Diego, CA, March/April, 1999.