Journal reference: *Computer Networks and ISDN Systems*, Volume 28, issues 7-11, p. 907.

Web Cache Coherence

Adam Dingle
KSVI
Charles University, Prague, Czech Republic
dingle@ksvi.mff.cuni.cz

Tomás Pártl
FJFI
Czech Technical University, Prague, Czech Republic
partl@km1.fjfi.cvut.cz

Abstract

All Web caches must try to keep cached pages up to date with the master copies of those pages, to avoid returning stale pages to users. In traditional distributed systems terminology, the problem of keeping cached pages up to date is called *coherence.* We discuss the coherence problem for Web caches, and argue that coherence techniques used for distributed file system caches may not be satisfactory for Web caches. We survey techniques used by popular Web caches for maintaining coherence, including the popular "expiration mechanism" which probably originated in CERN's proxy http server. We discuss a number of problems with the existing expiration mechanism, and present several extensions to it which solve these problems, reduce user wait times and decrease the staleness of returned Web pages. We also discuss *pre-fetching* and *replication*, more speculative techniques for keeping Web caches up to date.

Keywords: cache http proxy coherence

Introduction

Hierarchical Web caching has become a standard technique for decreasing latencies which users experience as they access Web pages. Caches provide decreased latency at a cost: every cache will sometimes provide users with *stale* pages - pages which are out of date with respect to their master copies on the Web servers where the pages originated. Web caching works because old information has value: stale information is often much better than none, and users are willing to tolerate receiving pages which are sometimes stale if they don't have to wait so long to receive pages. Unfortunately, as this paper will show, existing Web caches return stale information all too often, and still make users wait longer than is really necessary.

Every Web cache must somehow update pages in its cache so that it can give users pages which are as fresh as possible. Indeed, the problem of keeping cached pages up to date is not new to Web caches: after all, the Web is really just an enormous distributed file system, and distributed file systems have been with us for years. In conventional distributed systems terminology, the problem of updating cached pages is called *coherence*. As far as we know, no research paper has yet seriously addressed the issue of

Web coherence mechanisms.

In some ways, keeping Web caches up to date is easier than keeping distributed file system caches up to date, because distributed file systems often support distributed writes, whereas writes to a Web page can be performed only at the page's Web server. Still, if we are looking for a good cache coherence mechanism for Web caches, perhaps we should start by examining the coherence mechanisms used by popular distributed file systems. These coherence mechanisms can be divided into two general classes: *validation check* mechanisms and *callback* mechanisms.

Sun's NFS [Lyo85] [San85] [RFC1094], perhaps the first distributed file system to be widely used, uses a *validation check* mechanism to keep cached pages up to date. When a client machine reads a disk page from a server and caches the page, it marks the page with a timestamp. When the client uses the cached page at a later time, it performs a validation check: it sends the timestamp to the server to verify that the page has not been used since it was cached. To improve performance, a cache will perform this validation check for a given page at most once during a fixed short time interval, typically 3 seconds.

Newer distributed file systems such as Andrew [How88] [Kaz88], Coda [Sat90], and AFS [Tra96] use *callbacks*, a very different mechanism, to keep cached pages current. In these file systems, when a client receives a page from a server, it also receives a *callback promise* for the page - a guarantee that the server will notify the client if the page changes. If a client has not received a callback for a given page, then it knows that the page has not changed - provided that the client and server have not crashed since the callback promise was issued, and provided that network communication has remained troublefree. When a client crashes and then restarts, it contacts the server to find out whether the pages for which the client has callback promises have changed. Callback mechanisms remain somewhat vulnerable to network failures, and some distributed file systems which use callbacks have validation check mechanisms as well, which are used periodically to guard against the possible loss of callback promises.

All Web caches known to the authors use validation check mechanisms to maintain cache coherence, and the new mechanisms we propose in this paper use validation checks as well. There are serious problems with making callbacks work on the Web, which is orders of magnitude larger than other distributed file systems. A single page could be cached in thousands or even millions of caches (because each Web browser typically has its own local cache), and it is inconceivable that a server could contain a list of all these caches to notify when a page is changed. Of course, if caches are organized hierarchically, then perhaps a server would only need to notify a small number of top-level caches that a page had changed, and the change notification could be propagated through the cache hierarchy; but it would be much too expensive to perform such a propagation every time a page changed everywhere in the Web. Furthermore, the sheer size of the Web and its lack of centralized administration mean that it is inherently less reliable and more vulnerable to callback loss than other distributed file systems. What's more, implementing a callback mechanism would involve basic changes to the operation of the existing base of Web servers; such changes are generally not required for validation check mechanisms. For these reasons, we will not discuss callback-related mechanisms any more in this paper, and will leave them to further research.

The NFS validation check mechanism mentioned above was designed for local-area networks and is not scalable. We will see below that existing Web caches use a coherence mechanism which is similar to the NFS mechanism, but is more scalable when used with a hierarchy of caches. We will then show how this popular mechanism can be improved in several important ways.

Existing Coherence Mechanisms

HTTP headers used for coherence checks

HTTP [W3C96] defines several headers which were specifically designed to support caching. These headers will be of central importance to our discussion, so we will review their definition here. It should be noted that the HTTP specification specifies certain behaviors for Web caches, but does not specify how caches are to keep cached pages up to date.

The HTTP GET message is used to retrieve a document given its URL. It is clear from the HTTP specification that when GET is passed to a cache, the cache may choose to return a cached document; GET alone does not guarantee that it will return a fresh page.

The header "If-Modified-Since: *date*" can be appended to a GET message and changes the meaning of the GET: a page will be returned only if its last modification date is greater than the date in the If-Modified-Since header. A GET message with attached If-Modified-Since is referred to as a *conditional GET*. The HTTP specification is vague on the issue of a cache's reponse to a conditional GET: it doesn't clearly say whether a cache must always pass a conditional GET message up to the next-higher cache in the hierarchy, or whether a cache may respond that a document has not been modified if it has observed recently that the page has not been modified since the date specified in the conditional GET. The specification says that *"if the requested document has not changed since the time specified in this field the document will not be sent, but instead a Not Modified 304 reply."* A strict interpretation of this sentence would mean that a cache must always pass a conditional GET to the next-higher cache - but that would be unreasonable, because then every user request would result in a message to a page's home server, and then the servers of extremely popular pages would be inundated with hundreds of conditional GET messages every second. In practice, real Web caches interpret the specification more liberally, and often respond that documents have not been modified without contacting higher-level caches.

The header "Pragma:no-cache" can be appended to a GET message and indicates that a cache may not return a cached page; it must return an authoritative and definitely fresh version of the page retrieved from the page's home server. A "Pragma:no-cache" request passes up through all caches in a cache hierarchy to a page's home server. Most browsers offer a "Reload" button which retrieves a page using this header.

The header "Last-Modified:" is returned with every page returned from a GET message, and indicates when the page was modified to create the specific version being returned. A page's last-modified time acts as a unique version identifier for the page: if two caches contain copies of a page and those pages have identical Last--Modified: headers, then the contents of those pages are guaranteed to be identical.

The header "Date:" is returned in response to every GET request. (The HTTP specification is actually somewhat vague on the subject of whether Date: should be returned when a conditional GET returns no page, but experiments reveal that common Web servers return Date: even in this case.) Date: represents the last date at which a page was known to be good; this is not the same as a page's Last-Modified date. For example, suppose that user A requests a page from a proxy cache at 10:00; the proxy cache retrieves the page from its home server. If the page was last modified at 8:00, then the home server will send the page to the proxy cache with headers Last-Modified: 8:00 and Date: 10:00; the proxy will send the page to user A with the same headers. At 10:05, user B may request the same page from the same proxy cache. If the proxy cache decides to return its cached, possibly stale page rather than contact the home server, then the page sent to user B will contain the same headers Last-Modified: 8:00 and Date: 10:00. Date: contains extremely valuable information for the user, for it tells just how stale a page might possibly be; often, a page's Date: will be recent, meaning that the page's information is quite up-to-date even if the information in the page was created some time ago. As such, it is a real shame that Netscape, the most popular Web browser, does not provide any mechanism for displaying the value in a page's Date: header, even in the "Document Info" window.

Staleness

In much of what follows, we will talk about pages' *staleness*, which we will now define precisely.

Given a cached copy of a page, we define its *last-good time* as the last time at which the page's contents on its home server were exactly equal to the present contents of the cached copy. If the page has not changed on its home server since the cached copy was made, the page's last-good time equals the current time.

Given a cached copy of a page, we define its *staleness* as the time elapsed since its last-good time. If the page has not changed on its home server since the cached copy was made, the page's staleness will equal 0.

Naive coherence mechanisms

Several naive coherence mechanisms are in wide use by caches today, and are supported by the built-in caches of Netscape and other browsers. Check-Every-Time is a common naive mechanism. When a cache which supports this mechanism receives a GET or conditional GET message for a cached page, it always sends an conditional GET message to the next higher cache or server, passing the Last-Modified date of the cached page as the date in the If-Modified-Since header. It uses the response from the conditional GET message to determine whether its own cached copy is up to date and, in the case that the cache is reponding to a conditional GET message, whether a new page needs to be sent to the requestor. Check-Every-Time was perhaps the earliest cache coherence mechanism. It cannot be considered for serious usage today, because if all caches in a hierarchy used Check-Every-Time, every user page request would result in a message to the page's home server. Then home servers for popular pages would be inundated with conditional GET requests. Furthermore, a single conditional GET request can take seconds or even tens of seconds across the ever-crowded Internet, and it is unacceptable for users to experience such delays on every page request.

Never-Check is another mechanism supported by Netscape's cache. A cache which supports Never-Check will never send If-Modified-Since messages to check whether cached pages are valid. If all caches in a hierarchy use Never-Check, users will have to explicitly refresh all pages using "Pragma:no-cache" messages.

Expiration-based coherence

The proxy servers probably in widest use today - the World Wide Web Consortium's httpd [W3C96a] (formerly CERN httpd) and Netscape's Proxy Server [Net96] - use an *expiration mechanism* to keep their pages up to date. The details differ from server to server, but the underlying mechanism is the same for all servers. Each cached page is assigned an expiration time - a time for which it will be assumed valid. Any GET requests made before the expiration time elapses (including conditional GET requests) are answered by assuming that the cached page is still fresh. After the expiration time elapses, the cache will handle the first GET or conditional GET request for the page by sending a conditional GET to the next higher-level cache to check if the page has changed; at this time, the cache will reset the page's expiration time back to its default value.

The following pseudocode shows how W3C httpd 3.0 maintains cache coherence; it is derived from study of httpd's source code.

```
function GET (P:page, S: date)
// Answer a HTTP GET request.  S is the date in the request's If-Modified-Since: hea
// If there was no If-Modified-Since: header, S = -infinity.
// Note that the Date: and Last-Modified: headers are stored with every cached page,
// transmitted unchanged whenever a cached page is returned.
{
if (P in cache & current_time < cached_P.expiration_time)  // in cache, not expired
  {
  if (cached_P.last_modified <= S)
    return "Not Modified", Date: cached_P.date
  else
    return cached_P
  }

if(P in cache & current_time >= cached_P.expiration_time)  // in cache, expired
  {
  send If-Modified-Since(cached_P.last_modified) to higher-level cache
  if (response="Not Modified")
    {
    cached_P.expiration_time := new_expire_time(P)
    return "Not Modified"  // with headers, including Date:, as we just received the
    }
  else
    {
    cached_p := received_P
    cached_P.expiration_time := new_expire_time(P)
    return cached_P
    }
  }

if (P not in cache)
  {
  send If-Modified-Since(S) to higher-level cache
  if (whole document was returned)
    {
    cached_P := response
    cached_P.expiration_time := new_expire_time(P)
    }
  return response to client, including headers
  }
}
```

In the above pseudo-code, the function new_expire_time, which assigns a new expiration time for a cached page P, is undefined. Expiration-based caches use a variety of mechanisms to assign expiration times for cached pages. HTTP actually provides a header "**Expires:** date" which a server may return with a document, meaning that the document definitely will not change before the given date and probably will change soon after the given date. If a document contains an Expires: header, the expiration date in the header can be used directly as an expiration time for cached copies of the document. As Glassman points out [Gla94], the Expires: header is rarely used. This is not surprising, since the author of a WWW page usually can't estimate a document's lifetime at the time it's created. Only a small class of WWW documents (e.g. cultural listings, train schedules) have a natural expiry.

For documents with no Expires: header, the simplest expiration time algorithm simply assigns an expiration time equal to the current time plus a constant we will call "default expiration time". This constant seems to provide an upper bound on the staleness of pages which are returned, since pages which have been in the cache for longer than this amount of time will be checked on access. As we will see below, the "maximum staleness" is actually not a real guarantee when a hierarchy of caches is used.

Documents which have changed recently are more likely to change soon than documents which have not changed for a long time. As such, it makes sense to use a lower expiration time for recently-changed documents. Both W3C's and Netscape's caches can be configured to assign expiration times equal to the current time plus a constant (say, .20) multipled by the amount of time that has elapsed since a page was last modified. W3C httpd calls this constant the "LM_factor".

Problems with expiration-based coherence

Expiration-based coherence as implemented in W3's httpd and Netscape's Proxy suffers from several evident problems.

1. Users must wait for expiration checks to occur.

Whenever a cache that uses expiration-based coherence checks to see whether an expired document is up to date, the user is sitting staring at a blank Web browser page. Expiration checks are often passed up to top-level caches and may even take tens of seconds across a crowded Internet. We feel that these checks should be made only after sending the cached document to the user; if it turns out that a cached document has changed, it can be given to the user on the next request, or indeed on the same request if our proposed server-push mechanism is used. Because expiration checks translate directly into intolerable user delay, system administrators today typically configure proxy caches to use artificially high expiration times, e.g. 1 day. (On the popular UK National Cache at Hensa UNIX, an enormous Web cache which uses Netscape's Proxy Server, the expiration time is set to 12 hours.) This means that users frequently receive stale pages. On the current Internet we feel that much lower expiration times, e.g. 30 minutes or 1 hour, would be reasonable if expiration checks didn't force users to wait.

2. If a user is not satisfied with the staleness of a returned document, they have no choice but to use a Pragma:No-Cache request to reload the entire document from its home site.

With the existing coherence mechanism, the only way to get a document version which is fresher than the one which is returned from a GET request - or, indeed, to find out if there is a newer version than the one returned from a GET request - is to send a Pragma:No-Cache request, typically initiated by the user's pressing the Reload button in a Web browser. Pragma:No-Cache is a very expensive operation because it always forces a reload of an entire document from its home site. This limitation is particularly painful given that administrators set expiration times to be artificially high due to the first problem in this list, causing stale pages to be returned more often than would be necessary with an improved coherence mechanism.

3. The mechanism provides no strong guarantee as to document staleness.

The "default expiration time" mentioned above seems to provide an upper bound on the staleness of documents returned by a cache, but the upper bound actually does not hold when a hierarchy of caches is used. To see this, suppose that all caches in a hierarchy are set to use a default expiration time of 3 days. On October 10, cache C receives a document from a higher-level cache through a GET request; the returned document has a Date: of October 8, which means that it might have a staleness of up to 2 days. Cache C will set the expiration time for the document to October 13; if the page is requested from cache C on October 12, then C will not check to see whether the page has changed, and the returned page will have a staleness of 4 days. In general, if all caches in a hierarchy are configured to use a default expiration time of T, and the distance from a cache D to the top-level cache in the hierarchy is N (including both D and the top-level cache), then a page returned by D might actually have a staleness of up to N * T. We don't know whether Netscape Proxy Server has this problem; unfortunately, Netscape does not provide information about caching behavior in sufficient detail for us to adequately evaluate it.

4. Users cannot specify the degree of staleness they are willing to tolerate.

In the current expiration model, each cache can be configured with a default expiration time which provides something like an upper bound on the staleness of returned pages. It is unfortunate that this default expiration time is fixed for each cache, for users will differ in their tolerance for stale pages. It would be nice if each user request could contain a maximum allowed staleness for that request. In particular, that would allow a browser to offer a cheaper Reload function which would return a page guaranteed to be fresh without forcing a reload, solving problem (2), above.

5. When the user aborts a document load, caches often abort a document load as well.

Extremely often, users click on a hyperlink in a page before that page has finished loading; in this situation, the page which was loading vanishes from the user's screen, and the newly requested page begins to load. In this situation, it is important that the page which was loading continue to load into the user's local cache and into higher-level proxy caches, because it is likely that the user will visit the page again soon. Unfortunately, in this situation both Netscape Navigator's built-in cache and W3C httpd abort the page download, and Netscape's built-in cache even discards the portion of the page which has been downloaded. Users who are aware of this behavior become paranoid about clicking on hyperlinks before a page has fully loaded, because they want the loading page to be cached so that they won't have to wait for it to load again.

This is really a problem of robustness of implementation rather than a defect in existing cache coherence mechanisms. To be fair, this problem does not really affect Netscape's Proxy Server, which can be configured to continue loading a page in the background whenever at least 1% of the page has been loaded before the load is aborted.

New Coherence Mechanisms

To remedy the problems we've observed with existing Web coherence mechanisms, we propose three extensions to these mechanisms. Our three extensions are essentially independent.

Extension 1: Returning a stream of document versions in response to each request

Our first extension fixes problem 1, discussed above. In addition, it addresses a deeper problem related to staleness and delay.

In a hierarchical cache, there is a fundamental tradeoff between staleness and delay. Typically, low-level caches which can be accessed quickly from a user's site are smaller and less frequently accessed than high-level caches such as national or continental caches. If a page exists in a low-level cache, then, it is likely to be staler than the same page in a high-level cache, because the expected time since the page has been last accessed (and, possibly, updated) is greater. Existing caches must decide between delivering a stale page quickly or delivering a fresher page after a greater delay. This decision is difficult, because different users may have different tolerances for delay or staleness, and users' preferences in this matter may vary from page to page. We believe that that a cache should not make this decision at all and that indeed

- Upon requesting a page, a user should receive some version of that page as quickly as possible. As fresher versions of the page arrive, the page should be updated in the user's Web browser. (If some users find it too distracting to have the currently viewed page suddenly updated, perhaps users should have the option to have a dialog box pop up where they can specify whether they want to see a newly arrived page version).
- The user should always have a clear indication of the staleness of pages being viewed - for example, a text area in a Web browser which shows the date when the page being viewed was last known to be current.
- If a page being viewed has already expired (i.e. the current date is greater than the date in the **Expires:** header which the page was assigned by its server), then the user should be given a very clear warning that they are looking at old information - for example, a dialog box could appear on the screen with a message to that effect.

This behavior is based on the observation that in the Web, old information has value. Pages typically change gradually, and old information is better than none at all. Often, users can get the information they need from older versions of a page, and any user will prefer to have something to read than to stare at a blank page. In this respect the Web differs from "traditional" distributed file systems such as NFS and Andrew. Most Web pages are downloaded not by programs, but by people who are waiting to look at the pages to glean information from them. As a result, we feel that it's reasonable to say that the ordinary response to a request for a Web page is actually a series of pages which provide increasingly good approximations to the current version of the page (somewhat analogously to the way Netscape displays increasingly clear approximations of a JPEG image as it arrives). By contrast, a traditional distributed file system provides its services in the form of system calls such as open() and read(). Such a file system could be extended with calls which returned a series of versions of a given file, but it's hard to imagine that many applications would use these calls (in fact, the only such applications would probably be Web-browser-like applications).

Implementation

This extension is the most radical of our proposals. Fortunately, Netscape, the Web browser which is used for more than half of all Web requests, already supports a *server push* mechanism which allows a series of pages to arrive in response to a single request. This means that we can build proxy caches which implement our proposed behavior and that most users will be able to use these caches with no changes to their browser software. We hope that other browsers will adopt server push in the near future.

Server push is described in detail in the Netscape document An Exploration of Dynamic Documents [Net96a]. In response to a GET request from a client which supports server push, a server may return a series of document versions separated using MIME headers via the experimental MIME type "multipart/x-mixed-replace". The server may keep the TCP connection open if it wants to keep open the possibility of sending more document versions at a later time. The client browser will display each document version in turn, removing previous versions from the display. Some Web servers have used server push as a crude way of performing animation; newer mechanisms such as Java are much better for making animated Web pages. Still, server push is ideal for our needs since it allows a browser to display successively newer versions of a document as they become available. Note, however, that server push will probably work only for documents transmitted by HTTP, not for FTP or Gopher documents.

To implement document version streams, we extend the meaning of GET requests: any GET request, including a conditional GET, may now return a series of documents which contain successive versions of a Web page. Each document in the series will have its own Date: and Last-Modified: headers. In addition, we allow the possibility that a document in the stream is *null*. A null document carries only a Date: header and is used to indicate that the document previously returned in the stream was still valid at a date later than previously realized. If a null document arrives in a version stream before any non-null document has arrived in that stream, the Date: header of the null document indicates that the document whose date was passed in the If-Modified-Since: header was still valid at a date later than the date passed in the If-Modified-Since: header. A null document may arrive before a non-null document only in the case that the GET request was conditional. A version stream which contains only a single null document is equivalent to a Not Modified response to a conditional GET; recall that the Not Modified response also carries a Date: header. An example below illustrates how null documents are used in version streams.

Let us define the semantics of version streams more precisely. The response to a request of the form GET page If-Modified-Since: S will be a stream of versions V_1, V_2, ... V_n. Let D_i be the Date: of version V_i; D_i is defined for every version, because even null documents have dates. Then always S < D_1 < D_2 < ... < D_n. Furthermore, if the current time is T and the maximum staleness for the request is L (below, we define a new header which specifies the maximum staleness for every GET request), then D_n >= T - L. This means that the last version in a stream satisfies the maximum staleness requirement. After a stream of versions has passed through a set of caches, each cache will have a copy of the last version in the stream. Earlier versions exist merely for the user's benefit while the user waits for the last version to arrive.

We have developed a simple cache algorithm which allows caches to generate version streams. The algorithm is presented in our pseudo-code below, which also incorporates our other extensions to the expiration coherence mechanism.

Although Netscape supports server-push documents, it needs to be extended in several ways to work well with our version streams proposal. First, as mentioned above, Netscape currently provides no way for the user to see the value of the Date: header returned with a page. With version streams, it is especially important for users to be able to see Date: values so that they will know how stale each version in a stream is. Second, null documents in a stream will undoubtably confuse Netscape and other browsers. Ideally, a browser should react to a null document in a stream by updating the displayed Date with the Date value passed with the null document. Third, Netscape currently clears the browser window as soon as the first bytes of a new document in a server-push stream are read. This behavior will not work well for our version streams, because users will generally prefer to browse an older version of a document than to wait for a newer version to load. To work around these limitations, low-level proxies which receive requests directly from browsers could be programmed to optionally "fix up" streams before passing them to browsers. For now, such proxies could strip out null documents, which would solve the second problem listed above. They could also wait until an entire document version was read before sending it to the browser; as such low-level proxies will typically be connected to proxies along fast, high-bandwidth connections, this would solve the third problem listed above because users would effectively not have to wait for new versions to load. In the long run, however, version streams will work best if browsers are extended slightly to support them.

An example

An example will help to illustrate how null documents work in version streams. Suppose that a cache hierarchy consists of two caches, A (a higher-level cache) and B (a lower-level cache). Suppose that the current date is 6/1, and that the document http://nowhere.org/nothing has last been modified on 4/1. Cache A contains a cached copy of the document; in that cached copy, the Last-Modified: field is 4/1 (the cached copy is up to date) and the Date: field is 5/1 (A last knew the document to be good on 5/1). Furthermore, suppose that B contains an older cached copy of the document whose Last-Modified: field is 1/1 and whose Date: field is 3/1. Finally, a user's Web browser contains a cached copy whose Last-Modified: field is 1/1 and whose Date: field is 2/1. Let's consider the sequence of events that occur when the user visits the page.

First, the Web browser will send a conditional GET message to B with If-Modified-Since: 2/1. (Note that many existing caches would send If-Modified-Since: 1/1 in this situation. The difference is not extremely important; sending the current Last-Modified: value instead of the current Date: value merely means that the cache may receive some versions whose dates lie between the Last-Modified: value and the Date: value. We recommend that caches send the Date: value as an argument to If-Modified-Since: in this situation.) B will immediately send a null version with Date: 3/1 to the browser, indicating that the browser's cached version was still valid on 3/1. B will send a conditional GET message to A with If-Modified-Since: 3/1. A will send a version with Last-Modified: 4/1 and Date: 5/1 to B; B will pass this version down to the browser cache. Finally, A will send a conditional GET message to the server nowhere.org with Last-Modified: 5/1; the server will respond with a null version with date 6/1; this version will be passed down through the cache hierarchy to the browser, indicating that the previously sent version (with Last-Modified: 4/1) remains valid on 6/1.

Extension 2: Fixing expiration time computations to provide a real staleness guarantee

This extension fixes problem 3 listed above. The extension is quite simple, and can almost be considered a bug fix for the W3C httpd cache algorithm. As discussed above, the W3C cache computes an expiration time for each cached object by adding the current time to an expiration delay; the expiration delay is typically constant or computed as a fraction of the time since the page was last modified. This is essentially wrong; in the expiration time computation, the cached object's Date: value, which represents the last time at which the object was known to be good, should be used in place of the current time. Essentially, the expiration time represents the time at which the object will become too stale; the object's Date: value represents staleness which the object may already have.

Our pseudocode, below, contains this fix. In fact, our pseudocode does not bother to keep an expiration time for each cached object; it merely computes it dynamically by adding the object's Date: field to the maximum staleness allowed.

Extension 3: Allowing the user to specify a maximum staleness for each request

This extension fixes problems 2 and 4 listed above, and is also quite simple. As suggested before, it would be nice if users could specify a maximum acceptable staleness for every request; typically this staleness value will be set in a browser configuration dialog. We propose adding a new Maximum-Staleness: field to HTTP GET requests; this field contains a time measured in seconds, and represents the maximum acceptable staleness of the page returned by the request. When combined with our proposed extension 1 (version streams), the Maximum-Staleness: field represents the maximum acceptable staleness of the *last page* in the version stream.

Some people may object to allowing every user to set the Maximum-Staleness: field. Indeed, this field will directly impact the network bandwidth used by caches. If the users in a cache hierarchy use Maximum-Staleness: T for all requests and a page is popular, a top-level cache in a cache hierarchy will query the page's server approximately once every T seconds to find out if the page has been modified. Thus, very small values of T could have an adverse effect on network performance, effectively causing every page access to send an If-Modified-Since: message all the way up the cache hierarchy to the page's server. Very small values of T should be discouraged for ordinary page access. The authors intuitively feel that a value of T somewhere in the range of 20 minutes to 2 hours is reasonable for most users in the current Internet.

Pseudo-code for our extensions

The following pseudo-code implements all three extensions we have discussed. It differs somewhat in structure from the pseudo-code we presented above for current expiration-based caches. Note that the transmit() function is asynchronous, and returns immediately; the page or null message passed to transmit() is transferred to the requestor in the background (say, in another thread). Also note that if transmit() is called several times during the course of a single GET request, the transmitted pages are *not* intermingled as they are sent: a page specified in a call to transmit() will be sent only after all previous calls to transmit() have finished executing. Each call to transmit() results in a separate multipart/x-mixed-replace section of the HTTP response.

```
function update (P:page, S: date)
// Utility funcion for GET.  Tell the receiver information about the page P based on
{
  if (cached_P.date > S)  // we have newer information than he does
    {
    if (cached_P.last_modified > S) // we also have a newer page than he does
      transmit(cached_P)
    else // we must have the exact same page that he does, since his page was valid
      transmit(null, date: cached_P.date)  // tell him that his page was valid at ti
    }
}

function GET (P:page, S: date, M: relative_time)
// Answer a HTTP GET request.  S is the date in the request's If-Modified-Since: hea
// If there was no If-Modified-Since: header, S = -infinity.
// M is the time specified in the request's Maximum-Staleness: header.  If there was
// Maximum-Staleness: header, M is the default maximum staleness for this cache.

// Note that the Date: and Last-Modified: headers are stored with every cached page,
```

```
// transmitted as headers whenever a cached page is returned.

{
if (P in cache)
  {
    update(P, S);
    if (current_time < cached_P.date + M)  // the info we just sent satisfies the re
      return  // no need for more transmission
  }

// We have now done all we can to service the request using information in our own c
// Now generate a request for a higher-level cache.  In our request, we must use the
// copy of P, NOT the date S -- because it's possible that the requestor has newer i
// do, and thus that it will be necessary for us to update our own cache by download
// (which will not be passed on to the requestor).  In this situation, we may actual
// of versions, which could potentially waste network bandwidth if those versions ar
// the requestor.  Actually, it will be rare to receive many versions which are not

if (P in cache)
  our_date := cached_P.date
else
  our_date := -infinity

send GET(If-Modified-Since: our_date, Max-Stale: M) to higher-level cache

for (each version in response from higher-level cache)
  {
  if (reponse.body = null)  // new date for page we already have
    cached_P.date := response.date;
  else // actual page update
    cached_P := response;  // includes date and body
  update(P, S);
  }
}
```

More Speculative Techniques

The coherence extensions we proposed above ensure that a page which is found in a cache will be no staler than T + S, where T is the time which has elapsed since the page was last requested and S is the maximum staleness specified in the last request for the page. Usually, T will be much larger than S; we expect that S will be less than one hour, while on average T will be days. If a single user requests a page commonly, the page will stay in that user's local cache and it will always be immediately available with a staleness not significantly greater than the time since its last access. Similarly, if a community (university, state, country) requests a page very often, the page will be accessible from the community's cache with a staleness approximately equal to the time since last access. In both cases, the more popular a page is, the less stale it will be when retrieved. We expect that this coherence mechanism will provide better performance than existing mechanisms on cache hierarchies of any size.

Nevertheless, more speculative coherence techniques can be used to improve caching performance yet further. The coherence extensions we proposed above are not at all speculative, in that they never update a cache page before the page has been explicitly requested, and in that they never load a page into a cache before the page has been requested from that cache. On networks with high cost, it is probably best not to be speculative. If we do have enough network bandwidth to spend some of it downloading pages which might never be used, we might use two sorts of speculative coherence mechanisms: *pre-fetching* and *replication*.

Pre-fetching

A *pre-fetching cache* periodically refreshes cached pages so that they will be less stale when requested by the user. Our coherence mechanism could be extended to perform pre-fetching: for each page in the cache we could periodically send an **If-Modified-Since** message to find out if the page has been modified. If the page has not been modified, we update its last-good time, decreasing its potential staleness when requested by the user; if the page has been modified, we download a newer version of the page to the cache.

In designing a pre-fetching cache, the central problem we face is deciding exactly when to refresh each page in the cache. Intuitively, as a cached page ages, the chance that it will be used decreases, so it should be refreshed less often. If a page is old enough, it is probably not worth it to refresh it as all. More research is required to develop a mathematical model which can be used to determine optimal refresh intervals.

Replication

Replication refers to the distribution of popular pages to caches before those pages have ever been requested through those caches. Tim Berners-Lee has made a proposal [Ber95] which calls for research in replication. He argues that caching without replication will not be adequate for distributing popular Web pages - but we don't agree with his argument. Berners-Lee essentially gives three reasons why replication is necessary.

First, he argues that "the nesting of proxy server [sic] increases response time". This is true, but proposed extensions to HTTP which keep inter-proxy TCP connections alive for transferring multiple Web pages minimize this inefficiency. Even the mighty Netscape has or will very soon have support for keeping a TCP connection alive, as evidenced by the fact that Netscape 2.0 beta sends a "Connection: Keep-Alive" header with every request.

Berners-Lee's second point is that "unnested proxies will be very numerous and put a high load on orginal servers". This is also true, but as Web proxy hierarchies increase in power, users will have more and more incentive to connect through proxies, and low-level proxies will have more incentive to connect through higher-level proxies. Furthermore, the bottleneck today in downloading popular Web pages and archive objects is generally transmission across the Internet itself, not load on Web servers.

Thirdly, Berners-Lee claims that "cache hit rates tend to be under 50%", and cites Steve Glassman's paper describing experience with a moderately large proxy serving a community of users at DEC SRC. Low cache hit rates are often claimed for Web proxy caches, but we don't feel that these statistics are so damning at all. In a cache hierarchy all caches contribute together to generate a high hit rate for the user; each cache only receives the pages which lower-level caches did not have, and so its hit rate will only represent the pages which are serviced at a given level of the cache hierarchy. Also, we feel that proxy caches which are large enough to provide more convincing hit rates have perhaps not yet been built. We are confident that a continental cache which services, say, Europe or the west coast of the U.S., when combined with mid-level caches, will be able to provide a hit rate of 80-90% for pages which are missed by an organization's own cache such as a university cache. But even more fundamentally, the pages which Berners-Lee proposes replicating preemptively - popular pages such as Olympics results or comet pictures - are exactly those which *will* find their way very quickly to mid-level and low-level caches.

Despite all this, we are not ready to dismiss replication as an important extension to Web caching. If network bandwidth is cheap enough, replication will decrease delays and staleness for pages which users receive through the Web. We feel that the problem of whether replication is really necessary on today's Internet is still open, and that more experimentation will be necessary to come to a firm conclusion. If replication is performed, we feel that existing Web proxy servers should be used to perform it (and not, as Berners-Lee suggests as an alternative, NNTP servers).

A proposed mechanism for replication

Here is a proposed mechanism for performing replication through Web caches. We have not implemented this mechanism, and there are details to be filled in, but we hope that the mechanism can serve as a first proposal and a basis for further work.

In our mechanism, each Web caches preemptively sends popular objects to those Web caches which sit beneath it in the cache hierarchy. To do this, each Web cache maintains a list of lower-level Web caches which communicate with it. This list could be derived automatically by noticing which requests come from caches through observation of the **User-Agent:** HTTP header, or manually by having lower-level Web caches somehow register to say that they are interested in receiving replicated objects.

A Web cache will determine an object's popularity by counting the number of GET requests made by lower-level caches for the object. In making this tally, the Web cache will not count conditional GET requests, because we essentially want to measure the breadth of an object's popularity rather than its intensity of use at any one site. In fact, we will typically receive only a single non-conditional GET for a single object from a single lower-level cache if the object is popular at all, for it will remain in the lower-level cache and so further GET requests for the object will necessarily be conditional.

If the number of GET requests for an object is high, we preemptively send it to lower-level caches. Some lower-level caches may be more interested in receiving preemptively distributed pages than others; in particular, larger lower-level caches will have more interest in receiving pages because the probability that those pages will be requested by users served by those caches is higher. If each lower-level cache somehow registers its interest in preemptive distribution with the higher-level cache, perhaps each lower-level cache can specify a threshhold of popularity for objects to be preemptively sent to it. This threshhold could possibly be a percentage, say, 10%: then objects would be sent down to the lower-level cache if they had been requested by 10% of the lower-level caches served by the higher-level cache.

The preemptive distribution itself could actually be crudely performed without changes to HTTP. If a cache C wants to preemptively send a page to a cache D which sits directly below C in the cache hierarchy, C can contact D as if C were a client, sending an HTTP GET request for the page. If D does not have the page, D will send a GET message to C to download the page to its cache. (If D happens to have the page already, D will send a conditional GET request to C and no great harm will be done.) Some network bandwidth will be wasted if D now tries to send the page back to C. To prevent this waste, C may cut off the TCP connection of its GET request (ugh); if D is robust, it will finish downloading the page to its own cache anyway. As a prettier alternative, a new HTTP method or GET header could be defined which instructs a proxy to download a page to its own cache, but not to send it to the requestor.

Conclusion: The Future of Web Caching

It is our sincere hope that Web caching will replace all archives and mirrors on the Internet: each object will have a single name, and popular objects will be widely cached and quickly accessible. Despite the increased popularity of Web caches at the organizational and even national level, archives continue to be extremely popular, which unfortunately shows that there is still work to be done in Web caching. Undoubtably some of this work is essentially administrative and commercial: we hope that universities and companies will participate more and more actively in setting up high-level Web caches. Perhaps some high-level caches will even be commercially run; organizations with lower-level caches will pay for the privilege of connecting. But there is much research work to be done, too, of both a theoretical and experimental nature.

Theoretical work will involve constructing mathematical models of caching systems to answer basic questions. For a given network, how should the expiration time of cached objects be set to minimize cost? Here, cost can be computed as the sum of the several components. We can assign costs to delays which users experience, to the staleness of pages which users receive, and to the packets transmitted on a network. Should the expiration time depend on the time since a document has been modified, and if so, how? Quantitatively, how cheap does network transmission need to be for pre-fetching and replication to be worth doing? Just when should documents be prefetched, and exactly when should caches replicate popular documents by sending them to lower-level caches?

Experimental work is necessary because theoretical models can only help us so much in understanding the living, breathing Internet. We still don't really know whether a modified expiration mechanism is fundamentally adequate for providing fast access to the popular pages of the Internet through a multi-level cache hierarchy, or whether replication is really necessary as well. We need more experiments with extremely large caches with quantitative results showing average user delays so that we can try to account for those delays and minimize them. We appeal to Web cache researchers who have log data for large caches to make their data publicly accessible so that other researchers can run simulations with those logs; in those logs, IP numbers of host computers can be scrambled if there are privacy concerns.

Acknowledgement

The authors would like to thank AT&T for its generous support of this work through a research grant to Charles University.

References

[Ber95]
: Tim Berners-Lee (1995), "Propagation, Replication and Caching on the Web", http://www.w3.org/pub/WWW/Propagation/Activity.html

[Gla94]
: Glassman, S. (1994). A Caching Relay for the World Wide Web. Proceedings of the First International World Wide Web Conference. http://www.research.digital.com/SRC/personal/Steve_Glassman/CachingTheWeb/CachingTheWet
[How88]
J. H. Howard, M. L. Kazar, S. G. Menees, D.A. Nichols, M. Satyanarayanan, R. N. Sidebotham,

and M. J. Westl, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems,* 6/1, 1988.

[Kaz88]
Michael L. Kazar, "Synchronization and Caching Issues in the Andrew File System", *USENIX Conference Proceedings 1988*, pp. 27-36.

[Lyo85]
B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, D. Walsh and P. Weiss, "Overview of the Sun Network File System", Sun Microsystems Technical Report, January, 1985.

[Net96]
Netscape Communications Corporation, "Netscape Proxy Server", http://www.netscape.com/comprod/proxy_server.html

[Net96a]
Netscape Communications Corporation, "An Exploration of Dynamic Documents", http://home.netscape.com/assist/net_sites/pushpull.html

[RFC1094]
Sun Microsystems, "NFS: Network File System Protocol specification", RFC 1094.

[San85]
R. Sandberg, "Design and Implementation of the Sun Network Filesystem", *Proceedings of the USENIX 1985 Summer Conference*, 1985.

[Sat90]
M. Satyanarayanan et al, "Coda: A highly available file system for a distributed workstation environment", *IEEE Transactions on Computers,* 39/4, 1990.

[Tra96]
Transarc Corporation, "Transarc Product Information: AFS", http://www.transarc.com:80/afs/transarc.com/public/www/Public/ProdServ/Product/AFS/index.htm

[W3C96]
World Wide Web Consortium, "Hypertext Transfer Protocol", http://www.w3.org/pub/WWW/Protocols/

[W3C96a]
World Wide Web Consortium, "W3C httpd", http://www.w3.org/pub/WWW/Daemon/

About the authors

Adam Dingle earned a B.S.E. in Computer Science from Princeton University in 1990 and a M.S. in Computer Science from the University of California at Berkeley in 1992. Since 1994 he has taught computer science at Charles University in Prague. His interests in computer science include programming languages, distributed systems and networks, especially the Internet. His research presently focuses on distributed caching for the World Wide Web.


Tomás Pártl was born on May 27, 1974 in Prague. In 1992, he graduated from Southwestern Academy in San Marino, CA. Now he is a fourth year computer major at the Faculty of Nuclear and Physical Engineering of The Czech Technical University.