Chapter 12

Prefetching with HTTP

12.1 Introduction

Previous chapters have examined possible mechanisms to predict user actions on the Web for prefetching, and to evaluate proposed algorithms and implemented prefetching systems. However, even when we find mechanisms that work, challenges remain to get them fielded, notably limitations of the HTTP mechanism itself.

This chapter will provide some answers to the question of why prefetching is not presently widespread as well as why it should not be deployed within the current infrastructure, and what changes are needed to make prefetching possible on the commercial World-Wide Web.

Prefetching into a local cache is a well-known technique for reducing latency in distributed systems. However, prefetching is not widespread in the Web today. There are a number of possible explanations for this, including the following perceived problems:

- the bandwidth overhead of prefetching data that is never utilized,
- the potential increase in queue lengths and delays from naive prefetching approaches [CB98], and
- the effect on the Internet (e.g., added traffic and server loads) if all clients were to implement prefetching.

Fortunately, the first two objections can be satisfied by more careful approaches — prefetchers can be limited to high-confidence predictions, which can limit the bandwidth overhead, and prefetching systems can be used to moderate network loads when requests are spaced appropriately. The third is still up for debate, but when prefetching occurs

over a private connection for objects stored on an intermediate proxy (as in proxy initiated prefetching [FJCL99]), this is no longer an issue.

We will first define prefetchability and then review how Web prefetching has been suggested by many researchers and implemented in a number of commercial systems. In section 12.3 we will discuss a number of less-well-investigated problems with prefetching, and with the use of GET for prefetching in particular. In effect we argue that 1) GET is unsafe because in practice, GET requests may have (not insignificant) side-effects, and 2) GET can unfairly use server resources. As a result, we believe that prefetching with GET under the current incarnation of HTTP is not appropriate. We then propose extending HTTP/1.1 [FGM+99] to incorporate methods and headers that will support prefetching and variable Web server quality of service.

12.2 Background

While some researchers have considered prepushing or presending, this chapter is only concerned with client-activated preloading of content. We are less concerned here about how the client chooses what to prefetch (e.g., Markov model of client history, bookmarks on startup, user-specified pages, the use of proxy or server hints), with the understanding that a hints-based model (as suggested in [PM96, Mog96, Duc99] and others) has the potential to avoid some of the concerns discussed here but may also introduce other problems and likewise lacks specification as a standard.

Therefore, in this chapter we make the not-unreasonable assumption that a prefetching system (at the client or intermediate proxy) has a list of one or more URLs that it desires to prefetch.

12.2.1 Prefetchability

Can prefetchability be defined? Let us start with prefetching. A relatively early paper [DP96] defines prefetching as a cache that "periodically refreshes cached pages so that they will be less stale when requested by the user." While this definition does have a Web-specific perspective, it does not reflect current usage of the term since it is specific

to cache-refreshment. The more general definition from Chapter 1) is:

> *Prefetching* is the (cache-initiated) speculative retrieval of a resource into a cache in the anticipation that it can be served from cache in the future. (2)

Typically the resource has not yet been requested by the client (whether user or machine), but the definition applies equally to cache refreshes (when the content has changed). With this definition we can see that:

- Prefetching requires the use of a cache.
- Prefetchability requires cacheability.

Note that not all prefetched resources will be used before they expire or are removed from the cache. Some will never be requested by the client. This implies that:

- Prefetching requires at least idempotence, since resources may end up being fetched multiple times.
- Speculative prefetchability requires safety. Unless we can guarantee that the client will actually use this resource (before removal or expiration), a prefetching system should not perform any action that has unwanted (from either the client or server's perspective) side-effects at the server.

Therefore, now we have a definition for prefetchable:

> A Web resource is *prefetchable* if and only if it is cacheable and its retrieval is safe. (3)

Typical concerns for safety (e.g., those raised in RFC 2310 [Hol98]) are really concerns for idempotency, but idempotency is insufficient for prefetching, as the prefetched response might never be seen or used. Here we are referring to the definition of safe as having no significant side-effects (see section 9.1.1 of the HTTP/1.1 [FGM+99]).

12.2.2 Prefetching in the Web today

Prefetching for the Web is not a new concept. Discussions of mechanisms and their merits date back at least to 1994-1995.[1] Many papers have been published on the use of prefetching as a mechanism to improve the latencies provided by caching systems (e.g., [PM96, Mog96, WC96, CJ97, CY97, JK98, MC98, Pal98, SKS98, AZN99, CKR99, Kle99, Duc99, FJCL99, PP99b, SR00]).

A number of commercial systems used today implement some form of prefetching. CacheFlow [Cac02a] implements prefetching for inline resources like images, and can proactively check for expired objects. There are also a number of browser extensions for Netscape and Microsoft Internet Explorer as well as some personal proxies that perform prefetching of links of the current page (and sometimes bookmarks, popular sites, etc.). Examples include NetSonic [Web02], PeakJet 2000 [Pea02], and Webcelerator [eAc01b]. In addition, older versions of Netscape Navigator have some prefetching code present, but disabled [Net99].

12.3 Problems with Prefetching

Even with a proliferation of prefetching papers in the research community, and examples of its use in commercial products, there are a number of well-known difficulties with prefetching as introduced in the beginning of this section.

Some researchers even suggest pre-executing the steps leading up to retrieval [CK00] as one way to avoid some of the difficulties of content retrieval and still achieve significant response time savings. However, as suggested above, researchers and developers are still interested in the performance enhancements possible with speculative data transfer. When prefetching (and not prepushing/presending), the standard approach for system builders is to use the HTTP GET mechanism. Advantages of GET and standard HTTP request headers include:

- GET is well-understood and universally used.

[1] For one interesting discussion still accessible, see the threads "two ideas..." and "prefetching attribute" from the www-talk mailing list in Nov-Dec 1995 at `http://lists.w3.org/Archives/Public/www-talk/1995NovDec/thread.html`.

- There are supporting libraries to ease implementation (e.g., for C there is W3C's Libwww [Wor01] and for Perl there is Libwww-perl [Aas02]).

- It requires no changes on the server's side.

However, the use of the current version of GET has some obvious and non-obvious drawbacks, which we discuss in this section. Even from the prefetching client's perspective, there may be some drawbacks. In particular, just like users, a naive prefetcher may not know that the object requested is very large and will take minutes or hours to arrive.

In many cases, however, these drawbacks can be characterized as server abuses, because the inefficiencies and deleterious effects described could be avoided if the content provider were able to decide what and when to prefetch. In the rest of this section we detail the problems with prefetching, with emphasis on the consequences of using the standard HTTP GET mechanism in the current Web.

12.3.1 Unknown cacheability

One of the differences between Web caching and traditional caching in file and memory systems is that the cacheability of Web objects varies. One cannot look at the URL of an object and *a priori* know that the response generated for that URL will be cacheable or uncacheable. Some simple heuristics are known, including examining the URL for substrings (e.g., "?" and "cgi-bin") that are commonly associated with fast-changing or individualized content, for which caching does not provide much use. However, while a URL with an embedded "?" has special meaning for HTTP/1.0 caching (see the HTTP/1.1 specification [FGM$^+$99], section 13.9), a significant fraction of content that might be labeled dynamic (e.g., e-commerce site queries) does not change from day to day [WM00] and should be treated as cacheable unless marked otherwise (as stated by the HTTP/1.1 specification). Only the content provider will know for certain, which is perhaps why the cacheability of a response is determined by one or more HTTP headers that travel with the response.

Some researchers (e.g., [Duc99]) have proposed prefetching and caching data that is otherwise uncacheable, on the grounds that it will be viewed once and then purged

from the cache. However, the HTTP specification [FGM+99] states that such non-transparent operations require an explicit warning to the end user, which may be undesirable. Since many objects are marked uncacheable just for hit metering purposes, Mogul and Leach suggest something similar in their "Simple Hit-Metering and Usage-Limiting for HTTP" RFC [ML97] which states in part (*emphasis added*):

> Note that the limit on "uses" set by the max-uses directive does not include the use of the response to satisfy the end-client request that caused the proxy's request to the server. *This counting rule supports the notion of a cache-initiated prefetch: a cache may issue a prefetch request, receive a max-uses=0 response, store that response, and then return that response (without revalidation) when a client makes an actual request for the resource.* However, each such response may be used at most once in this way, so the origin server maintains precise control over the number of actual uses.

Under this arrangement, the response no longer has to be marked uncacheable for the server to keep track of views and can even direct the number of times that the object may be served.

In summary, though, when a client or proxy prefetches an object that it is not permitted to cache, it has wasted both client and server resources to no avail.

12.3.2 Server overhead

Even when the object fetched is cacheable, the server may have non-negligible costs in order to provide it. The retrieval will consume some portion of the server's resources including bandwidth and cpu, and logging of unviewed requests. Likewise, the execution of some content retrievals may not be desirable from a content-provider's point of view, such as the instigation of a large database search. In fact, the use of overly aggressive prefetchers has been banned at some sites [Pel96].

As more attention is paid to end-user quality of service for the Web (e.g., [BBK00]), the content provider may want to be able to serve prefetched items at a lower quality

of service [Pad95, PM96] so that those requests do not adversely impact the demand-fetched requests.

12.3.3 Side effects of retrieval

According to the HTTP/1.1 specification [FGM+99], GET and HEAD requests are not supposed to have a significance other than retrieval and should be considered safe. In fact, the GET and HEAD methods are supposed to be idempotent methods. In reality, many requests do have side-effects (and are thus not idempotent).

Often these side effects are the reason for some objects to be made uncacheable (that is, the side effects are desirable by the content producers, such as for logging usage). More importantly, some requests, when executed speculatively, may generate undesirable effects for either the content owner, or the intended content recipient. Placing an item into an electronic shopping basket is an obvious example — a prefetching system, when speculatively retrieving links, does not want to end up placing the items corresponding to those links into a shopping cart without the knowledge and consent of the user.

This reality is confirmed in the help pages of some prefetching products. For example, one vendor [eAc01a] states:

> On sites where clicking a link does more than just fetch a page (such as on-line shopping), Webcelerator's Prefetching may trigger some unexpected actions. If you are having problems with interactive sites (such as e-mail sites, online ordering, chat rooms, and sites with complex forms) you may wish to disable Prefetching.

Idempotency is a significant issue and is also mentioned in other RFCs (see for example [BLMM94, BLC95, BLFM98]). It is additionally an issue for pipelined connections in HTTP/1.1 (prefetching or otherwise) as non-idempotent requests should not be retried if a pipelined connection were to fail before completion.[2] The experimental RFC

[2]HTTP idempotency continues to be significant. For a discussion of HTTP idempotency with emphasis on idempotent sequences, see the discussion thread "On pipelining" from the IETF

2310 [Hol98] proposes the use of a new response header called Safe which would tell the client that the request may be repeated without harm, but this is only useful after the client already has header information about the resource. As suggested by Chapter 10, the lack of guaranteed safe prefetching also complicates live proxy evaluation.

The HTTP/1.1 specification [FGM+99] does note that "it is not possible to ensure that the server does not generate side-effects as a result of performing a GET request." A conscientious content developer would not only need to use POST to access all potentially unsafe resources but to prevent GET access to those resources, as linking to content cannot be controlled on the WWW. There may also be some confusion as to the interpretation of the idempotence of GET as a MUST rather than a SHOULD, and what constitutes safety and its significance (as a SHOULD requirement). In any case, enough content providers have built websites (perhaps without thought to prefetching) that do generate side-effects from GET requests that careful developers are unlikely to develop widespread prefetching functionality, for fear of causing problems.

12.3.4 Related non-prefetching applications

There are other situations in which server resources are potentially abused. These include price-comparison systems that repeatedly retrieve pages from e-commerce sites to get the latest product pricing and availability, and extensive mechanical crawling of a site at too fast a rate [Luh01]. Both of these can cause undue loads at a Web site and affect performance of the demand-fetched responses. While the need for an improved interface between crawlers and servers has been recognized (e.g., [BCGMS00]), the need for distinguishing between demand– and non-demand-requests has not been suggested. Interestingly, there has been work [RGM00] to build crawlers that attempt to access the "hidden Web" — those pages behind search forms [Ber00], further blurring the semantic distinction between GET and POST methods.

A server may also benefit from special handling of other non-demand requests, such as non-demand-based cache refreshment traffic [CK01b], in which additional validation

---

HTTP working group mailing list archives at `http://www.ics.uci.edu/pub/ietf/http/hypermail/2000/thread.html`.

requests are made to ensure freshness of cached content. From the content provider's perspective, it would be helpful in general to serve non-demand requests of any kind at a different quality of service than those requests that come from a client with a human waiting for the result in real-time.

12.3.5 User activity conflation

Non-interactive retrieval also has the potential for generating anomalous server statistics, in the sense that a page retrieval is assumed to be represent a page view. Almost certainly some fraction of prefetched requests will not be displayed to the user, and so the logs generated by the server may overestimate and likely skew pageview statistics. This is also the case for other kinds of non-interactive retrievals, such as those generated by Web crawlers. Today, when analyzing server logs for interesting patterns of usage, researchers must first separate (if possible) the retrievals from automated crawlers [KR98, Dav99c]. As Pitkow and Pirolli have noted [PP99a]:

> "... There is no standardized manner to determine if requests are made by autonomous agents (*e.g.*, robots), semi-autonomous agents acting on behalf of users (*e.g.*, copying a set of pages for off-line reading), or humans following hyperlinks in real time. Clearly it is important to be able to identify these classes of requests to construct accurate models of surfing behaviors."

Thus, if the server were maintaining a user model based on history, undistinguished prefetching (or other non-demand) requests could influence that model, which could generate incorrect hints that would cause additional requests for useless resources which would likewise (incorrectly) influence the server's model of the user, and so on. A server that is able to distinguish such requests from demand traffic would also be able to distinguish them in logs and in user model generation. Therefore, a mechanism is needed to allow a server to distinguish demand requests from non-demand requests, as describe below.

12.4 Proposed Extension to HTTP

The preceding section has discussed the problems with prefetching on the Web using current methods. In this section, we describe the need for an extension (as opposed to just new headers) for HTTP, mention briefly some previous proposals for HTTP prefetching extensions, and then provide a starting point for recommended changes to the protocol.

12.4.1 Need for an extension

HTTP is a generic protocol that can be extended through new request methods, error codes, and headers. An extension is needed to allow the server to define the conditions under which prefetching (or any other non-demand request activity) is allowed or served.

The alternative to an extension is more appropriate use of the current protocol. One approach, say, to eliminate some wasted effort would be to use range-requests (much like Adobe's Acrobat Reader [Ado02] does) to first request a portion of the document. If the document was small, then there is a chance the whole object will be retrieved. In any case, the headers would be available. A slight variation would be to use HEAD to get meta-information about the resource. That would allow the client to use some characteristics of the response (such as size or cacheability) to determine whether the retrieval is useful. These suggestions do not address the problems of request side-effects and high back-end overhead which would not be visible from headers, and in fact may be exacerbated by HEAD requests (e.g., if the server has to generate a dynamic response to calculate values such as Content-length). Neither do they allow for a variable quality of service at the Web server.

A second alternative would be to classify systems performing prefetching activity as robots, and require compliance with the so-called "robot exclusion standard" [Kos94]. However, conformity with this non-standard is voluntary, and is ignored by many robots. Further, compliance does not ensure a prevention of server abuse, as there is no restriction on request rates, allowing robots to overload a server at whim.

A third approach would be to incorporate a mechanism to support mandatory extensions to HTTP without protocol revisions. Such a mechanism has been proposed in Experimental RFC 2774 [NLL00], which allows for the definition of mandatory headers. It modifies existing methods by prefixing them with "M-" to signify that a response to such a request must fulfill the requirements of the mandatory headers, or report an error. If RFC 2774 were widely supported, this proposal might only need to suggest the definition of new mandatory headers that allow for prefetching and other non-interactive requests to be identified.

12.4.2 Previous proposals

Other researchers have proposed extensions to HTTP to support prefetching in one form or another.

- In his masters thesis, Lee [Lee96] prepared a draft proposal for additional HTTP/1.1 headers and methods. PREDICT-GET is the same as GET, but allows for separate statistical logging (so that logs are not skewed with prefetching requests). PREDICT-GOT is a method equivalent to HEAD, but is to be used as a way for the client to tell the server that the prefetched item was eventually requested by the client. Lee also lists new headers that would specify what kinds of predictions are desired by the client, and the kinds of predictions returned by the server.

- Padmanabhan and Mogul [PM96] suggest adding a new response header to carry predictions (termed Prediction) that would provide a list of URLs, their probabilities, and possibly their size. Elsewhere [Mog96], Mogul provides examples of other possible response headers that would be useful, including ones to describe mean interarrival times.

- Duchamp [Duc99] defines a new HTTP header (termed Prefetch) along with a number of directives. The new header is included in both requests and responses to pass information on resource usage by the client and suggested resources for prefetching by the server. The actual retrieval is performed using GET.

Unlike these proposals, we would like to see changes that are independent of a particular prediction technique.

12.4.3 Recommending changes

Any realistic set of changes will need to be agreed upon by a large part of the Web community, and so it makes sense to develop a proposal for these changes in consultation with many members of that community. However, there are certain aspects to the changes that appear to be sensible with respect to the type of prefetching described in this document.

We argue that a new method is needed. It should be treated similarly to the optional methods in HTTP/1.1. The capability of the server could be tested by attempting to use the new method, or by using the OPTIONS method to list supported methods. A new method provides safety — if a client uses a new method, older servers that do not yet support prefetching would return the 501 Not Implemented status code, rather than automatically serving the request, which would happen with the use of a new header accompanying a GET request. If the server does support these prefetching extensions, it would have the option to serve the request as if it were a normal GET, or at a lower-priority, or to deny the request with a 405 Method Not Allowed status code. When a client receives some kind of error code using the new method, it may choose to re-try the request with a traditional GET, but with the understanding that such action may have undesirable side-effects.

Therefore, we suggest a second version of GET (arbitrarily called `GET2`). A new header (possibly called `GET2-Constraint`) may also be useful to allow the client to specify constraints (e.g., `safe` to guarantee safety of the processing of this request, `max-size` to prevent fetching even a portion of an object that is too large, `min-fresh` to prevent fetching an object that will expire too quickly, `max-resp-time` to not bother if the response will take too long, etc.). Note that some of the `Cache-control` header values may be relevant here (like `min-fresh`), even though `GET2` requests may be made to servers as well as caches. Another header might be useful to allow client-type identification (e.g., interactive for browsers and other clients needing immediate response,

non-interactive for prefetching systems that will eventually send this content to users, robot for non-interactive systems that will parse and use content directly, etc.), but it may also be sufficient to assume that `GET2` requests will only be generated by non-interactive systems.

In summary, we recommend the development of a protocol extension that, when implemented by non-interactive clients that would otherwise use GET, allows for absolutely safe operation (no side-effects). When implemented by servers, such a protocol gives them the ability to distinguish between interactive and non-interactive clients and thus potentially provide different levels of service.

12.5 Summary

The benefits of Web cache prefetching are well understood. Prefetching could be a standard feature of today's browsers and proxy caches. This chapter has argued that the current support for prefetching in HTTP/1.1 is insufficient. Existing implementations can cause problems with undesirable side-effects and server abuse, and the potential for these problems may thwart additional development. We have made some initial suggestions of extensions to HTTP that would allow for safe prefetching, reduced server abuse, and differentiated Web server quality of service.

Overall, the conclusions we reach about the current state of affairs are the following:

- Prefetching with GET can be applied safely and efficiently between client and proxy under HTTP/1.1 using the `only-if-cached` and `min-fresh` options to the `Cache-control` header. Proxy resources can be abused, however, by a client that prefetches content that will never be used, and prefetching effectiveness is limited to the contents of the cache.

- Neither the safety nor efficiency of prefetching from an origin server can be guaranteed, as the origin server cannot recognize non-demand requests and so must serve every request, including requests for content that is uncacheable or has side effects.

- Proxies and origin servers cannot provide different qualities of service to demand and non-demand requests, again because they cannot distinguish between demand requests and non-demand requests (generated by prefetchers and Web robots).

- Much of the uncacheable content of the Web can be prefetched by clients in the sense that the content can be cached if semantic transparency is allowed to be compromised (although user warnings are still required under HTTP/1.1).

An extension to HTTP could rectify the second and third problems and allow for safe, semantically transparent prefetching for those systems which support it, and "gracefully" degrade to the existing situation if necessary when the extension is not supported by either the client or the server. It is our hope that this chapter will restart a dialog on these issues that will move in time into a standards development process.