Chapter 1

Web Proxy Caching

1 Introduction

1.1 Motivation

The World Wide Web is a large distributed system based on a client-server architecture. web clients request information from web servers which provide information through the network. The web community is growing so quickly that the number of clients accessing web servers is increasing nearly exponentially. This rapid increase of web clients makes the web slower. How can we reduce the latency web user's face when downloading web objects?

Several approaches have been proposed to increase the performance of the web. Trying to scale server and network bandwidth to keep up with increasing demand is one simple but expensive solution. Several observations suggest that a cache-based approach can help improve performance for lower cost. First, a single client often requests the same web object several times during a small interval of time. Second, web object accesses are non-uniform over all web servers; a small set of "popular" servers faces a disproportinate share of total server load. Further, different users often request the same web object from these highly loaded servers. If we can store commonly requested objects closer to web clients, users should see lower latency when browsing. web caches are the systems that keep copies of frequently accessed objects close to clients The development of web caching has spurned new research in many areas [7, 40].

1.2 Traffic Characteristics

The analysis of web traffic characteristics is important because specific access properties can be exploited to develop more efficient web caches.

Two of the most important characteristics of web traffic for web cache design are access frequency of a web object and temporal locality of reference in web request streams. It is a common belief that the frequency of web object access is not uniform for all web obejcts. Several early studies [3, 13, 18] have found that the relative frequency with which web pages are requested follows Zipf's law[44]. Zipf's law maintains that the relative probability of a request for the Ith most popular page is proportional to $1/I$. However, recent studies[33, 9] found that this distribution does not follow Zipf's law precisely, but instead follow Zipf-like distribution with a variable exponent.

The tendency that recently accessed objects are more likely to be accessed again in the near future represents temporal locality. Temporal locality of reference in web request

streams has been observed in many studies[5, 10, 20].

There is an inherent relationship between the temporal locality and skewed access distribution of web traffic characteristics. That is, the population is one of the major determinants of the temporal locality. However, there are other contributors to temporal locality, which is *temporal correlation* of repeated requests to the same web object[20, 22, 23].

The precise identification and characterization of web traffic properties can help improve the efficiency of web caches. Usually, workload characterization is done by analyzing traces of web traffic. Web workloads may be generated or recorded from the perspective of browsers, proxies, and servers. Workloads from clients can be valuable in evaluating user-level performance metrics and access patterns, but have limited value when looking at system-level performance metrics or the study of global properties.

Server workloads are often easy to find, as many servers log the requests they service. But workloads obtained from server logs do not reflect the access patterns of individual users. Thus, while they are useful when evaluating system-level performance metrics such as network bandwidth demand and server load, they are less effective when studying user-level performance metrics such as service time.

A caching proxy often functions as a second (or higher) level cache. That means that only the misses from web clients are passed to the proxy. Workloads from proxies usually do not exhibit per-user temporal locality of reference, but do track the traffic generated by a large number of user simulataneously.

1.3 Types of Web Caching

Information providers publish their information on the World Wide Web in a variety of formats. Usually information in the web is referred as *documents* or *web pages*. In terms of caching, unist of information in the web are not exactly simple documents or pages. Instead, a piece of a document or a part of stream information may be a unit of caching. So, we will use *web object* as a more generic term to represent a unit of web content.

Web caching is *managing copies of web objects closer to clients to enable clients to see lower latency when accessing to objects.* As a client requests a web object, it flows from a server, through a network, and to the client. Between a client and a server may be one or more proxy servers. There are three kinds of caches according to where copied objects are stored and managed.

1.3.1 Browser Cache

This type of cache is built on a web browser. A web browser supporting caching stores local copies of web objects which have been accessed based on a specific cache management policy. There are two forms of client caches[1]. A *persistent* client cache keeps web objects between invocations of the web browser. [1] A *non-persistent* client cache [2] removes cached copies when the user quits the browser.

1.3.2 Proxy Cache

In general, a proxy is a special HTTP server that can run on a firewall machine[32]. Proxying is a standard method for allowing accesses through a firewall without forcing each client to include customized support a special firewall environment. The cache is located on a machine on the path from multiple clients to multiple servers. Usually the same proxy is used by

[1] Netscape's Navigator browser uses a persistent cache.
[2] The NCSA Mosaic browser uses this type of cache.

all clients within a given subnet. This makes it efficient for a proxy to do caching of web objects requested by a number of clients. When a single proxy cache communicates solely with it's clients and servers, it is called an *isolated cache*. It is possible to use a set of caching proxies which cooperate with each other to improve performance. They are called *cooperative caches*. The configuration may be hierarchical so that the caches can be identified as first level caches, second level caches, and so on. It may be also non-hierarchical.

1.3.3 Server Cache

Server caching is another term for placing a cache in front of a web server. This is called "server" caching because it is implemented by the administrators of the web servers, rather than by the clients. The goal here is to cache and distribute web objects from the servers and to offload the processing of client requests.

1.3.4 Caching Hierarchy

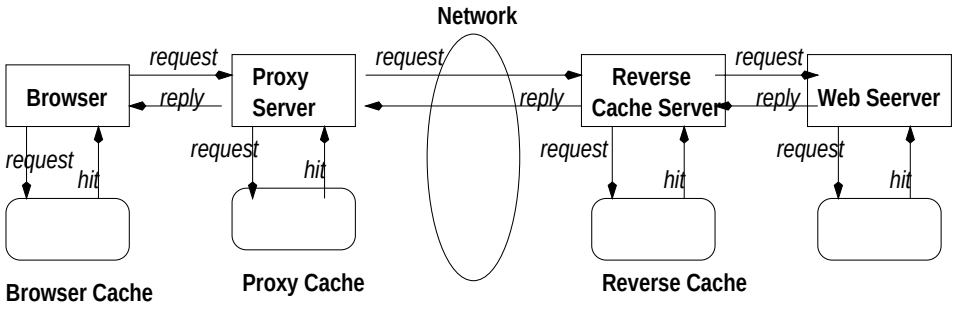

Figure 1.1: An example web cache hierarchy.

Hierarchical web proxy cache structures are similar to memory hierarchies[16]. Figure 1.1 shows an example caching hierarchy. At each level, requests are processed with cache misses propagating a request down the hierarchy one more level.

Web browser caches form a top of the hierarchy. By configuring browsers to direct web objects requests to a nearby proxy server, the proxy cache provides the second level of caching. The proxy cache then sees requests for all misses in the brower cache. Some proxy caches can be organized to act as a third-level caches as well. The lowest-level proxy cache is typically a cooperative cache that connects lower-level caches to each other so that a miss in one cache can be satisfied by one of its siblings. At the bottom of the hierarchy are the reverse caches which are closest to web servers. If requests for web objects are satisfied at higher levels of the cache hierarchy, then the cost of accesses can be significantly reduced..

1.4 Contrasting Web Cache Styles

A browser cache tries to reduce retrieval latency for a user by keeping their previously accessed web objects in a client's memory and disk. A browser supports the necessary functions for cache management including cache size, maximum retention period and mechanism to main consistency of copies and so on. For the user's perspective, reducing latency is the most significant factor over larger network efficiency considerations. These consideratons might conflict with the goals of an organization or network system. Some browser caching

provides an aggressive form of preloading of web objects which the user is likely to access. However, this may cause an unnecessary increase in network traffic.

Proxy caches lie in the middle of network and receive requests from many clients. So the requests seen by a proxy cache are interleaved across users. When requests are satisfied from a proxy cache, network traffic will be reduced because those requests need not be sent to servers. It also reduces client latency and server workload. As the proxy cache manages web objects requests from multiple clients, it is focused on improving global performance. Efficient management of cache space is more important in this case than for a browser cache. There are a number of issues to be considered for proxy caches and they will be discussed in the following sections.

Server caches are similar to proxy caches. However they accepting only traffic for their web servers so that they act as a true web server from the client's perspective. These caches are not for end users, but deployed and maintained for large web publishers. The focuses of server caches are primarily the scalability and server workload problems faced by popular web sites. Decreasing network traffic to web server is also a concern of these caches. It can also enhance security by forcing clients to communicate through anintermediate layer instead of directly with the true originating server.

Web caching needs to consider different issues compared to other caching systems in traditional client-server environments. Usually, traditional caching assumes that the data to be cached has a fixed size (i.e. pages or blocks). Time to load a page is also assumed to be fixed (i.e. main memory access time, or some i/o access time). But web objects exhibit large variations in size from a few hundred bytes to several of megabytes. Additionally, in the web environment, times needed to load web objects are variable and often difficult to predict. Further, web caching systems can still be useful with relatively weak consistency models - unlike a memory hierarchy, where weak consistency is wholly unacceptable. So, approaches for traditional caching do not fit for the problem of caching in a web environment. These differences make web caching an interesting and challenging problem.

1.5 Why Focus on Proxy Caches?

Proxy caches have been the subject of significant academic research[32, 18, 6] and also a significant area of commercial development. While an individual browser cache or server cache is beneficial to only a specific client or a web server site, a single proxy cache can benefit multiple clients and multiple servers at the same time.

A proxy cache has several potential advantages [18]. The first is that it can reduce latency on requests for cached pages because those requests need not be directed to original servers. As only missed requests from a proxy cache or explicitly requested ones to the server are sent to servers through the network, proxy caching can reduce both overall network load and server load. When a remote server is unavailable because of network disconnections or failures, cached copies are still abailable to users. However, proxy caching has several potential disadvantages. The basic problem associated with caching is the object returned to a user may be different from the originating server's current conent if the objects has changed at that server since the last cache update. Cache consistency mechanisms need to be applied to address this problem. When a request for an web object is made, it should be checked always whether the requested object exists in the cache. Otherwise, the request will be passed to the original web server. So deploying proxy caching may increase latency on requests for objects which are not cached. It also incurs a cost in administrative complexity, disk and memory space, and processing power. As a side effect of caching, objects hit counts in a server may not reflect users' real object access tendencies or true object popularity.

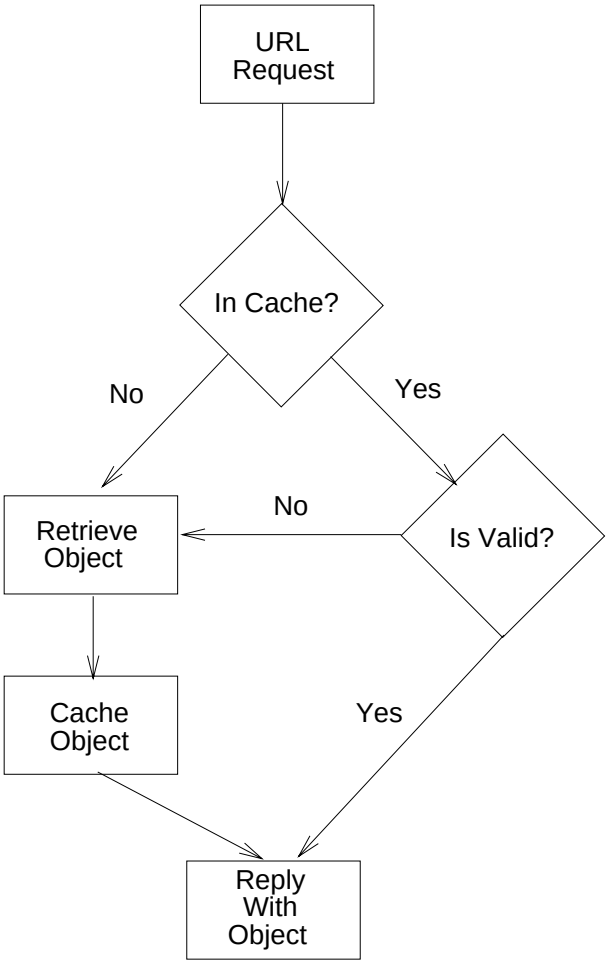

Figure 1.2: Basic flow diagram of a proxy cache.

2 How A Proxy Cache Works

The basic state machine implemented by a web proxy cache is depicted in figure Figure 1.2. User requests come into the cache as users browse the web; whatever happens within the cache, the result seen by the user is a returned web page. This user experience is captured in the top- and bottom-most blocks in the flow diagram: URL requests come into the system, and the system replies with pages. Performance is measured from the user's perspective, although there are a number of different measures proposed in the literature.

These user requests come into the proxy cache as HTTP commands. The user is presented with the view of a single proxy cache, but the cache may be internally distributed and repeated accesses may result in a different server processing the requests. Within the context of this flow diagram: the inflow of requests may be directed at a large number of "In Cache?" systems, each operating on a different computer.

When a request is received, the first operation performed determines whether the requested object has already been stored in the cache. This is simply a question of querying the contents of the cache, and returning a boolean result. Again, the cache may store objects

in a distributed fashion, resulting in a more complex operation to service this query.

If the object was found to be in the cache, the system then checks to see if the cached copy is "valid." The definition of valid varies from implementation to implementation; this is where the proxy cache designer can control the consistency model that is in place and the scope of objects that are considered cachable.

The weakest consistency model for a proxy cache always returns "Yes" at this point: all cached objects are considered valid. Similarly, the strongest consistency model always returns "No" at this point: the object is reloaded for every request. In between these extremes lie the consistency models of interest, using checks like the age of a cached object, time-to-live information from the server or dozens of other possible metrics. Valid objects are returned directly to the user.

The set of objects that are considered cachable is also defined at this stage. A proxy cache could rule out caching cgi content by always declaring any object with "cgi-bin" in the name to be invalid. A similar approach could insure that only HTML (and not ASP, AVI, etc...) content is cached.

If a cached object is invalid or if the object was not found within the cache, the system retrieves the data from the originating server. After the data has been downloaded, the system faces the task of caching it. This stage of the process includes the replacement policy component of web proxy cache design: when the cache does not have the space to store the downloaded object, it must evict objects from the cache to make room.

The policy used to select which objects are to be removed from the cache is user-specified. Further, the policy that determines when the cache is full is user-specified: the cache can impose restrictions on the number of cached objects from a single server or domain, or any other restrictions on cached objects the designer wants. Available cache space need not be the only consideration when deciding if replacement is needed, but it surely must be one of them.

Similarly, the system need not restrict the set of retrieved objects to those requested by the user - although the cache will need to retrieve the objects needed to satisfy user requests as they come in. A proxy cache could be built to retrieve (and then pass on to the "cache object" stage) documents without an explicit user request. This technique is known as "prefetching": downloading objects before an explicit request has been made for them. Whether to implement such a system, and what the list of prefetched objects should be, are both decisions left up to the proxy cache designer.

Within this overview of web proxy cache functionality, we highlight 6 issues for further discussion:

- proxy cache performance
- cache consistency model
- replacement policy
- prefetching
- cachability
- architecture

Each of these issues concerns the design of one or more steps of a proxy cache system, and has been the subject of significant academic and commercial research. Building a web proxy cache requires deciding how to handle each of these issues; building a proxy cache well-suited to your goals requires understanding each of them.

3 Quantifying Performance

3.1 Performance Metrics

Many different performance metrics have been proposed in the literature. We focus on 5 quantities that are widely measured and used by practitioners.

Hit Ratio is the number of requests that hit in the proxy cache as a percentage of total requests. When the cache size is small and the sizes of documents are similar, it will be a good measure.

Byte Hit Ratio is the number of bytes that hit in the proxy cache as the percentage of the total number of bytes requested. When the cache size is big and the sizes of documents are different, it will be a good measure.

Latency Reduction is the percentage of the sum of downloading latency for the pages that hit in cache over the sum of all downloading latencies. When hte waiting time has become the primary concern of Web users, it will be a good measure.

Hop Reduction is the ratio between the total number of the hops of cache hits and the total number of the hops of all accesses. When the network cost is mainly from the number of network hops traveled by documents, it will be a good measure.

Weighted-Hop Reduction is the corresponding ratio for the total number of hops times "packet savings" on cache hits. A cache hit's packet saving is 2 + filesize/536, as an estimate of the actual number of network packets required, if the request is a cache miss (1 packet for the request, 1 packet for the reply, and size=536 for extra data packets, assuming a 536-byte TCP segment size). When the network cost is proportional to the number of bytes or packets, it will be a good measure.

3.2 Performance of various replacement policies

3.2.1 Hit Rate and Byte Hit Rate

Maximizing the Hit Rate and Byte Rate is one of the major concerns for proxy caching. GreedyDual-Size (GD-Size) (1) and GD-Size(packets) are two of replacement algorithms that can achieve this goal. They are two versions of the GreedyDual-Size algorithm. The cost for each document is set to be 1 to minimize miss ratio in GDSize(1) and the cost for each document is set to be 2+size/536 to minimize the network traffic resulting from the misses in GD-Size(packets).

Simulation tests show that GD-Size(1) can achieve the best hit ratio among Least Recently (LRU), Size, Lowest Relative Value (LRV), GDSize(1) and GD-Size(packets). It performs particularly well for small caches. Thus it would be a good replacement algorithm for the main memory caching of web pages.

However, since GD-Size(1) considers the saving for each cache hit as 1, regardless of the size of document, GD-Size(1) achieves its high hit ratio at the expense of a lower byte hit ratio GD-Size(packets), on the other hand, achieves the overall highest byte hit ratio and the second highest hit ratio. GD-Size(packets) tries to minimize overall network traffic.

LRV outperforms GD-Size(packets) in terms of hit ratio and byte hit ratio. Because many workloads have significant skews in the probability of references to different sized files, and LRV knows the distribution before-hand and users it to improve performance. However, for all other traces where the skew is less significant, LRV performs worse than GD-Size(packets) in terms of both hit ratio and byte hit ratio.

Above all, for proxy designers that think hit ratio is most important, GD-Size(1) is the appropriate algorithm. On the other hand, GD-Size(packets) is the appropriate algorithm to achieve both a high hit ratio and a high byte hit ratio.

3.2.2 Latency Reduction

As the waiting time to retrieve a web object has become the primary concern of web users, reducing the latency of HTTP requests has become a major concern for proxies. Hybrid is one replacement cache replacement algorithm which takes into account the different latencies incurred in loading different web pages, and attempts to minimize the average latency seen by a system user. This algorithm has a lower average latency than LRU, LFU and SIZE.

The two versions of GreedyDualSize that take latency into account are GD-Size(latency) and GD-Size(avg latency). The cost of a document is set to be the latency that was required to download the document in GD-Size(latency). The cost of a document is set to be the estimated download latency of a document for GD-Size(avg latency).

GD-Size(1) performs the best, yielding the highest latency reduction, among LRU, Hybrid, GD-Size(1), GD-Size(latency) and GD-Size(avg latency). GD-Size(latency) and GD-Size(packets) finish the second. And LRU follows close behind. GD-Size(avg latency) performs badly for small cache sizes, but performs very well for relatively large cache sizes. And last, due to its low hit ratio, Hybrid performs the worst. Hybrid has a low hit ratio because it does not consider how recently a document has been accessed when making replacement decisions.

In summary, GD-Size(1) is the best algorithm to reduce average latency.

3.2.3 hop Reduction and Weighted-hop Reduction

Network cost is another concern for web proxy caches. GD-Size(hops) and GD-Size(weightedhops) are two replacement policies that incorporate network cost. In GD-Size(hops) the cost of each document is set to be the hop value associated with the Web server of the document, and in GD-Size(weightedhops), the cost is set to be hops*(2 + filesize/536).

Simulation shows that algorithms that consider network costs do perform better than algorithms that ignore them. For hop reduction, GD-Size(hops) performs the best, and for weighted-hop reduction, GD-Size(weightedhops) performs the best. This shows that GreedyDual-Size is also very flexible and can accommodate a variety of performance goals.

Thus, GD-Size(hops) is a good choice for the regulatory role of proxy caches. On the other hand, GD-Size(weightedhops) is the appropriate algorithm if the network cost is proportional to the number of bytes or packets.

3.2.4 Summary

Based on the above results, GD-Size(1) is the appropriate algorithm to achieve high hit ratio or low average latency. If a high byte hit ratio is also important in the proxy, GD-Size(packets) is a good choice. If the documents have associated costs (i.e. network transmit time) that do not change over time, or change slowly over time, then GD-Size(hops) and GD-Size(weightedhops) are the appropriate algorithms to consider.

4 Cache Consistency

4.1 Overview

Though web proxy caching provides several beneficial effects, it introduces a new problem not present in cacheless web environment. As there can be more than one copy of a given web object, a user might see an old copy of a requested object when the cache returns its data but the originating server has changed the object since the cache last copied it. For web caches to be useful, cached copies should be updated when the original objects change. When cached copies are not up-to-date, they are considered to be "stale". Cache consistency mechanisms insure that cached copies of objects obey certain rules - a cache consistency policy - with respect to their being out of date[19, 15].

Current consistency models for web cache maintenance can be divided into two categories. Strong consistency is the model which ensures that a stale copy of data will never be returned to a user. However, no cache's implement the strong consistency model because of the unbounded message delays in the Internet and the limited utility of a strong-consistent proxy cache. Weak consistency is a broad class of models in which a stale data might be returned to the user. So, weak consistency may not always be satisfactory. Users should be aware that the cache might occasionally return a stale data.

In order to maintain strong consistency, a proxy cache must know exactly when the original objects change in order to reflect those changes in its own copies. However, there is no way for the cache to know when objects change without either asking the server or providing a mechanism whereby servers can inform caches about changes. For the weak consistency case, a proxy cache should determine whether a copied object should be considered as up-to-date or not. A proxy cache can use some information from the server alongside its own decision algorithms to estimate the validity of the copied data more accuractely. Such estimation might not always be correct, resulting in the cache occasionally returning stale data.

Cache consistency algorithms have been extensively studied in the context of traditional distributed file systems and client/server database systems. Most of the traditional algorithms guarantee strong consistency and require servers to maintain state about the data cached by clients. However, the web is fundamentally different from a distributed file system and a client/server system in its access patterns[19]. Also, the scale of the web is orders of magnitude larger than any distributed file system, making these schemes intractable for web cache consistency. As changes for an object are made from a single web site, caches should never observe conflicting update instructions, and that may make the consistency issues simpler. Such different environments may make the techniques used in conventional systems not be adequate for caching on the Web.

4.2 HTTP mechanisms to support cache consistency

HTTP[8, 15] defines several headers which were specifically designed to support caching. Though the HTTP specification specifies certain behaviors for web caches, it does not specify how to keep cached objects up to date.

The HTTP GET message is used to retrieve a web object given its URL. However GET alone does not guarantee that it will return a fresh object. HTTP headers that may effect caching can be classified into two categories. The first category includes headers appended to retrieve a web object for cache control. The second category includes headers appended when a web object is returned.

4.2.1 HTTP Headers Appended to a GET Message

- *If-Modified-Since:date* : When appended to a GET message, a web object will be returned only if its last modification date is greater than the date in the If-Modified-Since header which is the last time a cache obtained a new copy from the originating server. Otherwise, "a Not Modified 304 reply" is returned. A GET message with attached If-Modified-Since header is called a conditional GET.

- *Pragma:no-cache* : When appended to a GET message, it indicates that a cache may not return a cached object. Instead, it must return a fresh version of the object retrieved from the object's home server. Most browsers offer a "Reload" button which retrieves an object using this header.

4.2.2 HTTP Headers Returned in Response to the GET Message

- *Expires:date* : This header notifies caches of the time until the object should no longer be considered fresh. After that time, every request for the object should be sent to the originating server to see if the object has changed. Expires headers will be specially effective for web objects for which it is relatively easy to estimate expirations. For example, static images which don't change much or objects that tend to change regularly are be good candidates for this approach.

- *Last-Modified:date* : This header returns the time the object was last modified to create the current version. This header is mandatory. Every object returned from a GET message will have this header. The last-modified time of an object can be a unique version identifier for the object. If cached objects have identical *Last-Modified:date* headers then the contents of those objects are guaranteed to be identical.

- *Date:date* : This header provides the last time an object was considered to be fresh. This is not the same as an object's Last-Modified date. This may inform users how stale an object might possibly be. For example, when an object's *Date:date* is recent, then it means the object's information is quite up-to-date even if the content of the object was created long before. So, this header reveals very important information for users.

4.2.3 Headers in HTTP 1.1 for Cache Control

While the "Expires:date" header can support control of caching to some extent, the HTTP 1.0 protocol does not provide much support for controlling when an object is cacheable or how a cache can manipulate it. HTTP 1.1 provides a new class of headers that makes it possible to define how caches should handle different web objects.

Some of the interesting options for *Cache-Control: options* response headers are as follows.

- *max-age=[seconds]* : This header specifies the maximum duration when an object may be considered to be fresh. This option supersedes the *Expires* header. Clients may send this header in order to explicitly and flexibly specify the degree of staleness acceptable to the user. Note that *no-cache* Pragma directive could only be used to flush caches unconditionally.

- *s-maxage=[seconds]* : This header specifies the max-age only for proxy caches.

	Strong Consistency	Weak Consistency
Client-based	Check-every-time	Never-check Expiration-based Piggyback-based Validation
Server-based	Invalidation-based Lease-based	Piggyback-based Invalidation

Table 1.1: Category of Cache Consistency Maintenance Approaches

- *public* : This header marks the response as cacheable regardless of whether it should be cacheable or not. Using this header, even an authenticated web object will be cacheable.
- *no-cache* : This header forces caches (both proxy and browser) to submit the request to the originating server for validation before releasing a cached copy every time.
- *must-revalidate* : This header asks caches to strictly obey any freshness information given for an object. HTTP allows caches to determine whether they will use the given freshness information or not. However, if this header is specified, caches should follow the freshness information for an object, and without modification.
- *proxy-revalidate* : This header is similar to "must-revalidate" but it only applies to proxy caches.

HTTP 1.1 also introduces a new kind of validator called *Etag* which is a unique identifier generated by the server and changed when the object changes. If the ETag matches in the response to a If-None-Match request, the object's content is the still the same.

4.3 Metrics for the Degree of Consistency

- degree of freshness/ staleness
- performance metric

4.4 Approaches for Cache Consistency

In order to keep consistency, several approaches have been proposed in the literature. They provide a spectrum of consistency levels based on the consistency guarantees provided by each mechanism. Further, different approaches interact with different parts of the caching system, which can lead to different consistency guarantees. In a client-based approach, the requests that maintain consistency are initiated by a client. In order to keep certain degree of consistency, it is necessary for clients to preform certain actions or to provide certain information to servers. For a server-based approach, the requests that maintain consistency are initiated by the server.

Table 4.4 shows the spectrum of several approaches. We will now describe the basic mechanisms of the approaches in each category.

4.4.1 Client-Based Strong Consistency Mechanisms

Check-Every-Time

This approach requires that proxy caches ask the server every time to determine if the data has changed. Thus, clients will never get stale data. Specifically, when a cache receives a GET or conditional GET message for a cached object, it always sends a conditional GET message to the next higher cache or server by passing the Last-Modified date of the cached object as the date in the *If-Modified-Since header*.

Check-Every-Time is one of the simplest consistency algorithms. The advantage is that it does not require any state to be maintained at the server, nor does the server need to block on a write request since the responsibility of maintaining cache consistency lies with the proxy. However, this approach has a large message overhead because it has to communicate with the originating server for every object request. It also increases the response time since the proxy waits untill it gets the result of every check.

Because of its high message overhead, this mechanism is rarely used for a consistency mechanism in current proxy caching systems. However it can be used as a part of a consistency mechanism for web objects which are uncacheable([3].

4.4.2 Client-Based Weak Consistency Mechanisms

Never-Check

This is the simplest consistency mechanism. It will never try to determine the freshness of a cached object without an explicit request for a validity check. Specifically, a cache will never send If-Modified-Since messages to check whether cached objects are valid or not. The responsibility of cache consistency lies solely on a client. In order to access a fresh object, clients will have to explicitly refresh the object using "Pragma:no-cache" message.

Expiration-based

This approach assumes that a cached object remains valid for a *Time-To-Live(TTL)* period, which is an *a priori* estimate of how long the object will remain unchanged after a client validates the object[19]. Current web caches consider a cached copy valid until it's TTL expires. Any GET requests made before the TTL of the corresponding object expires will return the cached objects by assuming those objects are still fresh. For requests on objects whose TTL are expired, GET or conditional GET(*If-Modified-Since* message) requests are sent to the upper level cache or the originating server to check whether those objects have changed.

With this approach, choosing the appropriate value of the TTL represents a trade off. If it is long enough, it will reduce validations for the number of object requests. On the other hand, a long TTL increases the likelihood that a cache will return stale objects.

The adaptive TTL(also called the Alex protocol[11]) handles the problem by adjusting the TTL duration of an object according to observations of the lifetime of the object. It takes advantage of the fact that object lifetime distributions tend to be bimodal, which is based on the assumption that young files are modified more frequently than old files and that the old files are less likely to be modified. This implies that validity checks for the older objects can be made less frequently. With adaptive TTL, a cache manager assigns a TTL value to an object, which is a percentage of the object's current age(i.e. current time minus the last modified time of the object).

Expiration-based approaches are now the most widely used But stale objects can still be returned to users.

[3]The circumstances when an object is uncacheable will be discussed in the cacheability issues in Section 7

Piggyback-based Validation

This approach is based on piggybacking cache state information onto HTTP requests to servers [25, 27]. Whenever a proxy cache communicates with a server, it piggybacks a list of its cached objects from that server. These cache copies of these objects might be stale, and the purpose of the exchange is to determine object-by-object which need are actually stale. The possibly stale objects are either objects with unknown expiration time or with expired TTLs. Then the server replied to the proxy cache with a list of which cached objects on the list are stale. The cache can update its data to remove any stale objects. Then a relatively short expiration duration (i.e. one hour) threshold is set at the proxy cache. If any access on a cached object is made during this duration, it is considered as fresh. Otherwise, the object is validated using a *IF-Not-Modified* request to the originating server.

Piggyback-based validation does not require any new connection between proxies and servers. However the proxy mnust maintain a list of cached objects for each server, and a server must process validation requests from caches. The performance of this approach depends on the number of requests from a proxy to a server and the number of objects cached at a proxy for a particular server. If there are few requests from a proxy server to a server, then chances for the cached objects to be validated will decrease greatly. A piggyback-based validation approach will then devolve into a check-every-time strong consistency mechanism. When there are many such requests the cache contents are validated at the granularity of the time duration. In this extreme case, this approach is like an expiration-based weak consistency approach.

4.4.3 Server-Based Strong Consistency Mechanisms

Invalidation-Based

Invalidation is based on servers notifiying caches of object changes [30]. In order to do this, servers keep track of which clients are caching which objects. A server notifies the clients with copies, and receives acknowledges from the clients before any modifications.

This approach is optimal in the number of control messages exchanged between servers and the proxies. However it may require a significant amount of state to be maintained. When there are a large number of clients, this approach suffers from burdening the server with a large bookkepping load. In addition, a server might send invalidation messages for clients that are no longer caching corresponding objects. The read cost is low because a client is guaranteed that a cached object is valid until told otherwise. However, when an object is modified, the server must invalidate the cached objects - so the write cost is high. Furthermore, if a client has crashed or if a network partition separates a server from a client, then a write may be delayed indefinitely. A study[30] shows that overhead for an invalidation-based approach is be comparable to the weak consistency approaches based on expiration.

Lease-Based

Invalidation-based approach require a significant amount of state to be maintained while expiration-based approach impose a large control message overhead. Lease-based approaches try to support strong consistency while providing a smooth tradeoff between the state space overhead and the number of control messages exchanged.

In lease-based approaches [42, 17], the server grants a lease to each request from a proxy. A lease is an associated timeout duration in which a server guarantees to provide invalidation

for modified objects. To read an object, a client first acquires a lease for it. The client may then read the cached copy until the lease expires. When an object is modified, the object's server invalidates the cached objects of all clients whose leases have not expired. To read the object after the lease expires, a client first contacts the server to renew the lease. The duration of the lease determines the server and network overhead. The smaller the lease duration, the smaller the server state space overhead, but at the cost of a larger number of control messages exchanged and vice versa. When lease duration is infinite, this approach reduces to an invalidation-based approach. When lease duration is zero, this approach reduces to an expiration-based approach.

Strong consistency can be maintained for server failures in the lease-based approach. If a client or network failure prevents a server from invalidating a client's cache, the server need only wait until the lease expires before performing the write. By contrast, invalidation-based approaches may force the write to wait indefinitely. Leases also improve the scalability of writes. They enable a server to contact only recently active clients (those holding leases on the object) rather than contacting all clients that have ever read the object.

4.4.4 Server-Based Weak Consistency Mechanisms

Piggyback-Based Invalidation

This approach is based on piggybacking. The server replies to proxy requests with the requested object and a list of modified objects from the list of objects that proxy has previously retrieved.

To improve the efficiency of these algorithms, servers and proxies exchange caching information at the level of volumes - collections of web objects. Servers partition the set of objects at a site into volumes, either a single site-wide volume or related subsets of objects. Each volume has a unique identifier and a current version. When a server receives a request from a proxy client containing the client's last known version of the volume, it piggybacks a list of objects in that volume that have been modified since the client-supplied version. The proxy client invalidates cached entries on the list and can extend the lifetime of entries not on the list.

Servers maintain volume, but no proxy-specific information. Whenever an object changes within a volume, the server updates the volume version and records the object that changed between the previous and current versions. Each proxy client maintains the current set of server volume identifiers and versions for the objects in its cache. When a proxy needs to request an object from a server, it looks up the current volume identifier and version for the object and piggybacks this information as part of the request. If the volume identifier is unknown or if the proxy does not have a version for the volume, then it requests such information to be piggybacked in the reply.

In response, the server piggybacks the volume identifier, the current volume version and a list of objects from this volume that have changed between the proxy-supplied and current version. The proxy client updates its volume version, uses the list to invalidate cached entries from this volume and possibly extends the expiration time for volume resources that were not invalidated.

When this approach is combined with piggyback cache validation, it is reported to provide nearly strong cache coherency with a staleness ratio of 0.001 and a 6-9% reduction in overall costs in comparison to the best TTL-based policy[28]. However, this mechanism requires changes to existing web servers for implementation.

5 Replacement Policies

5.1 Why We Research Replacement Policies

A cache server has a fixed amount of storage for storing objects. When this storage space is full, the cache must remove some objects in order to make room for newly requested objects. The cache replacement policy determines which objects should be removed from the cache. The goal of the replacement policy is to make the best use of available resources, such as disk, memory space and network bandwidth. Since web use is the dominant cause of network backbone traffic today, the choice of cache replacement policies can have a significant impact on global network traffic.

5.2 Factors to be Considered

5.2.1 Live Documents

We say a document is live if that document will be requested in future. The cache only needs to retain live documents to achieve the maximum hit rate. Live documents are a small fraction of all documents. Thus it is more appropriate to consider documents to be dea if they have not been requested for more than some reasonably large time.

5.2.2 Interaccess time

Interaccess time is the time between sucessive document requests. Documents having lower interaccess times are the documents that are more likely to be requested in the future. Due to always selecting the document with the largest interaccess time to be evicted, the LRU algorithm is the best replacement algorithm for reducing average cached-object interaccess time.

5.2.3 Number of Previous Accesses

Using the number of previous accesses made to a document is a good indication. We can use it to evaluate whether the document will be requested in the future. However, since it does not include any aging information about the document, this cannot be used alone as the deciding factor.

5.2.4 Document Size

The document size is another important factor for caching. In proxy caching the cached documents can be of different sizes. Having more documents in the cache will likely lead to a higher hit ratio, so one might choose to cache more small documents at the expense of performance for larger documents.

5.2.5 Type of Document

The type of the document can also be an important factor to consider. Actually, many of the requested objects are rather small image files, suggesting that a bias for document type could be beneficial.

5.2.6 Latency

It is also important to consider the cost inccured in acquiring the document. The more expensive the document to download, the better it is to retain the document in the cache because the penalty for a cache miss is greater.

5.3 Existing Replacement Algorithms

Existing replacement algorithms are classified into three categories, according to whether they exploit access recency and access frequency, and whether they are sensitive to the variable cost and size of objects.

5.3.1 Recency-Based Policies

The Least Recently Used algorithm (LRU) [14] is the most widely used cache replacement algorithm, as it captures recency and is superior to other simple policies like FIFO and Random. Since Web traffic exhibits temporal locality of reference, LRU is widely applied in Web servers, client applications, and proxy servers. A disadvantage of LRU is that it does not consider variable-size or variable-cost objects.

The LRU-MIN [1] algorithm is a policy derived from LRU that tries to minimize the number of documents evicted by applying LRU only to the documents whose size is above some threshold. The threshold is adaptive: if there is not enough space left, the threshold will be lowered and the policy reapplied.

The GreedyDual-Size (GDS) algorithm [10] is an algorithm that can achieve the best overall performance by considering locality, size and latency/cost and combining them effectively. GDS is a variation on a simple algorithm named GreedyDual (GD) [43], which deals with uniform-size variable-cost objects. It assigns a value H to each cached page p. At first, when a page is brought into cache, H is set to be the cost of bringing the page into the cache. When a replacement is needed, the page with the lowest H value, min_H, is replaced, and then all pages reduce their H values by min_H. If a page is accessed, its H value is restored to the cost of bringing it into the cache. Thus, the H values of recently accessed pages retain a larger portion of the original cost than those of pages that have access. This algorithm integrates the locality and cost concerns very well.

A common drawback of LRU and GreedyDual-Size is that they do not take into account the frequency of resource use.

5.3.2 Frequency-Based Policies

The basic frequency-based replacement algorithm is Least Frequency Used (LFU) [14]. It always removes the object with the lowest reference count. LFU is online-optimal under a purely independent reference model. However,there are two subtle problems with LFU. First, there are different versions of LFU algorithm, such as Perfect LFU and In-Cache LFU, according to whether the reference count is also discarded when an object is evited. Second, in an LFU replacement algorithm, when two objets have the same reference count, a tiebreaker is necessary.

Server-weighted LFU (swLFU) [24] is a simple generalization of LFU. It permits servers to increase the allocation of shared cache space to the URLs they host, thereby reducing server workloads. Weights in swLFU represent the extent to which servers value cache hits, and swLFU is sensitive to differences in server valuations. Lots of simulation results demonstrate that under a particular artificial assignment of valuations to servers in actual

trace data sets, swLFU delivers higher aggregate value to servers than LRU or LFU, and furthermore can provide reasonable variable QoS to servers.

Hybrid [41] algorithm is aimed at reducing the total latency. It not only considers the connection time of a server and the network bandwidth that would be achieved to the server, but considers document size and number of document access. And it combines them in a efficient way. A function is computed for each document which is designed to capture the utility of retaining a given document in the cache. The document with the smallest function value is then removed. The function for a document located at server depends on the following parameters: the time to connect with server, the bandwidth to server, the number of times the document has been requested since it was brought into the cache, and the size (in bytes) of the document. Performance Experiments show that Hybrid is a robust policy. When using download rate along with other factors, Hybrid and SIZE are superior for HR, LFU and LRU.

5.3.3 Recency/Frequency-based Policies

- Fixed Cost/Fixed Size Algorithms

  Several studies have considered both recency and frequency information under a fixed cost/fixed size assumption.

  The LRU-K[34] algorithm is to keep track of the times of the last K references to popular database pages, using this information to statistically estimate the interarrival time of such references on a page by page basis. Many simulation results prove that the LRU-K algorithm has significant cost/performance advantages over conventional algorithms like LRU, since LRU-K can discriminate better between frequently referenced and infrequently referenced pages. Moreover, unlike the approach of manually tuning the assignment of page pools to multiple buffer pools, this algorithm is self-reliant in that it does not depend on any external hints.

  The LFU-DA algorithm [4] is a frequency-based algorithm with dynamic aging. On a fetch or a hit, the object value is set to the reference count plus the minimum reference count in the cache. LFU-DA calculates the key value k_i for object i using the following equation: $K_i = C_i * F_i + L$,with C_i set to 1. This equation uses only the frequency count and the inflation factor to determine the key value of an object. Simulations with large traces indicate LFU-DA obtains the highest byte-hit-ratio. Furthermore, the LFU-DA policy may be useful in other caching environments where frequency is an important characteristic but where LFU has not been utilized due to cache pollution concerns.

  The Least Recently/Frequestly Used (LRFU) [29] policy is a new block replacement policy that includes both the LRU and LFU policies, depending on the different weights given to recency and frequency. Simulation results show that if the cache size is large enough to hold most of the working set, such as the case where the cache size is larger than 200 blocks for our workload, the point near the LFU extreme on the spectrum gives the lowest miss rate. moreover, When the cache size is 350 blocks, the LRFU policy gives about 30% better miss rate than the LRU policy. This superior performance of the LRFU policy results from the fact that it considers the frequency factor as well as the recency factor when it decides the block to be replaced.

- Variable Cost/Size Algorithms

  To deal with variable cost/size, generalizations of the above techniques have also been proposed.

In [39], the Least Normalized Cost Replacement algorithm for proxy caching on the Web (LNC-W3) is proposed as a generalization of LRU-K to deal with variable-cost and variable-size Web objects. It is a delay-conscious cache replacement algorithm which explicitly consider the Web's scale by preferentially caching documents. It computes the average reference rate and uses that to estimate the profit of caching an object. Simulation indicated that LNC-W3 obtains higher delay saving ratios than those achived through LRU and LRU-K.

In another direction, since GD-size policies do not take into account how many times the object was accessed in the past, several studies proposed generalizations of the GreedyDual-Size algorithm to incorporate frequency. These algorithms include GreedyDual-Size-Popularity (GDSP) [21] policy, GreedyDual-Size with Frequency policy (GDSF) [4] and greedyDual-Least Frequestly Used (GD-LFU) [26] algorithm.

GDSP is a generalization of GDS that enables it to leverage the knowledge of the skewed poplarity profile of Web objects. It incorporates access frequency into the GDS algorithm. A popularity profile of Web objects requested through the proxy is maintained efficiently, which makes it possible to accurately estimate the long-term access frequency of individual objects. This algorithm can exploit temporal locality exhibited in the Web traffic as well as avoid cache pollution by previously popular objects. Trace simulations indicate that when HR is the main objective, GDSP is the best choice. It outperforms GDS without significantly compromising BHR. As to latencysaving ratio for NLANR traces under LRU, LFU, GDS and GDSP, the results show that latency reduction is minimal for LRU and LFU. But GDSP clearly outperforms GDS.

Lowest Relative Value algorithm (LRV) [31] includes the cost and size of a document in the calculation of a value that estimates the utility of keeping a document in the cache. The algorithm evicts the document with the lowest value. The calculation of the value is based on extensive empirical analysis of trace data.

Among all documents, LRV evicts the one with the lowest value. Thus, LRV takes into account locality, cost and size of a document. The performance simulation of LRV, compared to other algorithms,such as LRU, LFU, size and FIFO shows that LRV features a consistently higher BHR than other policies in all conditions. The same happens for the HR, except in the case of the SIZE policy with large caches. But reducing the cache size causes the SIZE policy to worsen because of the pollution of the cache with small documents, which are never replaced. LRV is perticularly useful in the presence of small caches.

6 Prefetching

Prefetching is a technique that use the prediction of the user's future access to retrieve data and thus help to reduce the user's perceived latency. By studying some web proxy trace, Kroeger et. al. [Kroeger97] found that local proxy caching with unlimited cache size could reduce latency by at best 26by at the best 57could provide at the best a 60boundary was only derived from limited traces, it showed the potential of using prefetching to improve the performance of caching. However, the benefit user perceived latency saving by prefetching comes with the cost of increasing network traffic and server workload, thus studying the tradeoff between performance and bandwidth saving is important in prefetching. For proxy cache, the space balance between caching and prefetching need also to be studied carefully when employ prefetching.

6.1 Prefetching technique classification

6.1.1 By Location

Prefetching can happen between web servers and proxy caches or between proxy caches and browser caches. In the first case, proxy cache act as client, prefetching web documents in local cache. In the second case, proxy cache act server, providing web objects for client prefetching.

6.1.2 By Information Resource

The information used for prediction algorithm can either come form the statistics of access history or from the accessed objects themselves.

Prefetching using the statistics from the history information can can be further classified into server based prefetching, local based prefetching, and hybrid prefetching.

- **Server based** In server based prefetching, the information for prediction is gathered by the server. The server use access history from a lot of clients to make the prediction. The server can either push the web objects need to be prefetched to the client, or give the client the prefetching information and let the client to decide what to prefetch. Proxy caches can either act as a server or act as a client here depending where the prefetching happens.
- **Local based** In the local based prefetching, the client use it own access history to make predictions and send request to the server. The client here can be a bower cache using one user's history or a proxy cache using a lot of user's access history.
- **Hybrid** In this approach, the predictions from the server and from the client are combined for prefetching.

The accessed web objects can also be the information resource for making prediction. For example, the hyperlinks in HTML pages can be the candidates for prefetching.

6.1.3 By Content

- **Object prefetching** The web object itself is prefetching based upon prediction.
- **Connection prefetching** While prefetching objects might increase the network traffic and server load dramatically, an alternative of prefetching connections is brought

up as an compromise. Cohen et. al. [Cohen99] proposed host-names pre-resolving (pre-performing DNS lookup), pre-connection (prefetching TCP connections) and pre-warming (sending "dummy" HTTP HEAD request) techniques to reduce user perceived latency. Their trace-based simulations show that connection prefetching has better performance improvement per bandwidth than object prefetching.

6.1.4 By Execution Time

- **Immediate prefetching** Prediction and prefetching is conducted immediately after each access.
- **Delayed prefetching** The bandwidth usage due to HTTP traffic often varies considerably over the course of a day, requiring high network performance during peak periods while leaving network resources unused during off-peak periods. Maltzahn et. al. [Maltzahn99] proposed "bandwidth smoothing" technique to use the these extra network resources to prefetch web content during off-peak periods. Their result showed that this technique can significantly reduce peak bandwidth usage without compromising cache consistency.

6.2 Prediction Algorithms

6.2.1 Prediction by Partial Matching(PPM)

PPM algorithm and it's variations are used by a lot of researchers as the prediction algorithm for prefetching[cao98, Papadumata96, Palpanas98 Foygel99]. This algorithm keeps track of the sequence of l accessed objects following a sequence of m objects. The data structure is typically a collection of trees. For prediction, the past up to m references are matched against the collection of trees to produce sets of objects as the prediction of the next l steps. Only candidates whose probability of access are higher than a certain threshold are considered for prefetching.

6.2.2 Top-10

Top-10 approach proposed in [Markato96] combines the servers' active knowledge of their most popular documents (there Top-10) with client access profiles. Based on these profiles, clients request and servers forward to them, regularly, their most popular documents.

6.2.3 Date Mining

This algorithm is used in [Aumann98]. The access workload is divided into a set of sequence. The support of s' in a sequence S is defined as the number of times S' appears in S as a subsequence. The support of S' in the whole training set X is the sum of it's support in sequences in X. Then all the frequent subsequence with support higher than a threshold can be computed from the training data. For prediction, given a sequence $S = (e_1, ..., e_m)$, consider all possible extensions of sequence $S' = (e1, ...e_m, e)$ for all values of e. For all suffixes of each extension, the extensions with greater support and longer matches are given a higher weight. The prediction is the extension with the highest weight.

6.2.4 Interactive Prefetching

For each user accessed HTML page, prefetch all its referenced pages.

6.3 Conclusion

Prefetching can be viewed as a technique to improve the efficiency of caching by a studying the access history. The precision and efficiency of the prediction algorithm are both very important in the performance of prefetching. If the prediction of the algorithm is imprecise, it will lead to too much network traffic. If the algorithm require too much computation in each prediction step, it might be unpractical to be used in reality. The integration of caching and prefetching is also an important problem need to be studied in prefetching.

7 Cacheability

The complexity of web objects makes cacheabilty unique problems in web caching. Recent study shows that a fair amount of web objects in the web accessing workloads are unrealistic to be cached in proxy cache[...]. The existence and the amount of uncacheable web objects make them the biggest reason of the degrading of the performance of web caching. Some research has been deployed to cache certain kind of web objects that are usually considered uncacheable otherwise. However, the intricacy and diversity of the web objects makes a universal solution to the cacheablity problem impossible. The lack of coordination between web server and proxy caches also makes this problem very complicated. Most of current work only stays at the research level.

7.1 What Kind of Objects are Usually Considered Uncacheable

Web objects can be considered uncacheable by the proxy cache for different reasons. Here is some major consideration.

First, some web objects are uncacheable by nature. For example, web objects that requires authentication upon retrieval shouldn't be cached. Some web objects are user specific, or context specific, which means that the result of the request depend on who is requiring it or the context of the requirement when it is made. This kind of web object should not be cached also since proxy cache usually don't make decisions (return cached copy or not) according to individual user and context. This kind of web objects are usually web objects including cookies.

Second, there are some web objects that can be cached in the proxy cache, however, no or little benefit can be got by caching those web objects. Thus, these kind of objects are also considered uncacheable by the proxy cache. They include the objects that are changing too fast and the objects that are too large. If an object is too dynamic, then the cached copy will be stale very soon, and need to be fetched from the server again upon the next request. Thus it is no good for the proxy cache to maintain them in the cache at all. A lot of dynamic generated web objects tend to change very fast. It's also hard to estimate the expiration time for a dynamic generated object in the adaptive TTL consistency policy. Thus dynamic generated web objects are usually considered uncacheable by the proxy caches. If an object is too large, then caching it will cause the eviction of then the reloading of a lot of small objects, which degrade the performance of the cache over all. Thus, a lot of proxy caches put a threshold on the size of cacheable web objects.

Finally, there are some web objects set to be uncacheable by the servers due to some reasons although they are cacheable to the proxy caches. For example, some web servers want to get the real access statistics for advertisement, so they don't want their web pages to be cached by the proxy caches.

7.2 How to Decide Which Web Objects are Uncacheable

It is the proxy who decide whether a web object is cacheable to it or not. However, the information they used to make the decision is very get from the web servers. Here is the source of the information that the proxy caches usually used to decide the cacheability web objects. Detailed description can be found in [zhang].

URL Dynamic generated objects can usually be identified from the URLs of the requests. Their URL always include "?", "=", "/cgi-bin/", ".cgi", ".pl" or ".asp".

HTTP header The HTTP response header contains the following information always imply that the object is uncacheable.

- pragma: no-cache
- Authorization
- Cache-Control: no-cache / private / no-store
- Set-Cookie
- No Last Modified Date or the Last modified Date is the request time.
- size: above threshold

HTTP status codes A lot of HTTP status codes imply that the response is uncacheable. For example, a response with code "302 Moved Temporarily" with no expire date is uncacheable.

7.3 Uncacheable Objects in Proxy Cache Workloads

A lot of researchers study the cacheability on the proxy cache workloads. The result depend on the workloads they studied and the different consideration of what kind web object is uncacheable. A brief summarization of these researches from 1997 to 1999 can be found in [Wolman99]. Although the total percentage of uncacheable objects under different consideration in different workloads may vary form 15they all imply that cacheabililty has a big influence on the effectiveness of caching.

7.4 Improving Cacheability

Since the web objects are uncacheable due to different reasons, making some uncacheable objects cacheable need also to be done according the specific reasons. There are some research work proposed to cache dynamic generated objects in the proxy caches. Streaming caching is a technique to cache large multimedia files in proxy caches. These works are still staying at the research level.

7.4.1 Caching Dynamic Henerated Content

Douglis et. al. [Douglis97] extended HTML to allow the explicit separation of static and dynamic portions of a resource. The static portions can then be cached, with dynamic portions obtained on each access.

Cao et. al. [Cao98] propose the Active Cache scheme to support caching of dynamic contents at Web proxies. The scheme allows servers to supply cache applets to be attached with documents, and requires proxies to invoke cache applets upon cache hits to finish the necessary processing without contacting the server. The user's access latency is saved at the expense of the proxy caches CPU costs.

Smith et. al. [Smith99] propose Dynamic Content Caching Protocol, to allow individual content generating applications to exploit query semantics and specify how their results should be cached and/or delivered. They classify locality in dynamic web content into three kinds: identical requests, equivalent requests, and partially equivalent requests. Identical requests have identical URLs, which result in the generation of the same content. The URLs of equivalent requests are syntactically different but the result generated by these requests are identical. Partially equivalent requests are syntactically different. Results generated for partially equivalent requests are not identical but can be used as temporary place holders for

each other while the real document is being generated. Dynamic Content Caching Protocol allow individual content generating applications to specify equivalence between different GET-based requests they serve. This information can be used by web caches to exploit these additional forms of locality. Identical requests and equivalent requests can directly fulfilled using previously cached results. For partially equivalent requests, previously cached content can be immediately delivered as an approximate solution to the client while actual content is generated and delivered. This allows clients to browse partial or related or similar results promptly while waiting for more accurate information.

7.4.2 S

tream Caching Realtime streams such as video are several orders of magnitude larger than normal web objects. This limits their cacheability in the ordinary proxy caches. However, the increasing demand of video and audio streams makes stream caching of particular interest in proxy caching.

Sen et. al. [sen99] propose prefix caching, that, instead of caching entire audio or video streams (which may be quite large), the proxy should store a prefix consisting of the initial frames of each clip. Upon receiving a request for the stream, the proxy immediately initiates transmission to the client, while simultaneously requesting the remaining frames from the server. In addition to hiding the latency between the server and the proxy, storing the prefix of the stream aids the proxy in performing workahead smoothing into the client playback buffer. By transmitting large frames in advance of each burst, workahead smoothing substantially reduces the peak and variability of the network resource requirements along the path from the proxy to the client. They construct a smooth transmission schedule, based on the size of the prefix, smoothing, and playback buffers, without increasing client playback delay. It's showed that a few megabytes of buffer space at the proxy can offer substantial reductions in the bandwidth requirements of variable-bit-rate video.

Rejaie et. al. [Rejaie99] present a fine-grain replacement algorithm for layered-encoded multimedia streams at Internet proxy servers, and describe a pre-fetching scheme to smooth out the variations in quality of a cached stream during subsequent playbacks. This enables the proxy to perform quality adaptation more effectively and maximizes the delivered quality. They also extend the semantics of popularity and introduce the idea of weighted hit to capture both the level of interest and the usefulness of a layer for a cached stream. It is showed that interaction between the replacement algorithm and pre-fetching results in the state of the cache converging to the optimal state such that the quality of a cached stream is proportional to its popularity, and the variations in quality of a cached stream are inversely proportional to its popularity. This implies that after serving several requests for a stream, the proxy can effectively hide low bandwidth paths to the original server from interested clients.

8 Web Proxy Cache Architecture

8.1 Disk vs. Memory Based Caching

The first issue to consider when designing a web proxy caching architecture is how to store the cached data. Caches can be implemented with disk, solid state memory, or any other data storage technique; most web proxy caching systems are implemented using a mixture of disk and fast memory.

Why is the storage question important for web proxy caches when we do not consider it at all for memory hierarchies? The basic answer is that web caches store a lot more information than memory hierarchy caches. While there are many implemented multi-gigabyte web proxy caches, caches within a memory hierarchy rarely make it above the 1-8 megabyte range. Building multi-gigabyte caches using only fast memory is sigificantly more expensive than using disk and can limit total capacity. Using disk, the amount of caching space is virtually unlimited. Using memory, we face the normal restrictions regarding very large memory systems [...].

Most users are familiar with the hybrid memory-disk approach taken by browser caches. These systems tend to work well and help reduce WWW latency. On the other size, server-side caches are normally implemented using only memory since the pages themselves are available on disk for the server. Proxy caches lie between these two tools on the network, and the storage requirements represent a combination of the two as well.

Disk has the clear advantage of nearly unlimited cache size. But in exchange for this scalability we loose latency with respect to memory-based systems. Disk bandwidth can also be orders of magnitude lower than memory bandwidth. On the other hand, memory is very quick both in terms of access time (latency) and bandwidth.

There are clear tradeoffs here. Having a larger cache should increase the hit rate, thereby increasing performance. But having a cache that itself introduces less latency should also increase performance. It is not clear which of these effects will results in better performance; most likely this result will vary from situation to situation.

8.2 Distributed Cache Systems

The basic web proxy cache is implemented on a single computer, located somewhere on the WWW. For many large-scale web proxy cache systems this model is inadequate. For a variety of reasons, these systems often employ a collection of computers in a distributed fashion. There are three main motivating factors here: increased cache size, increased compute power to process requests and the ability to be “closer” to more clients by spreading the cache out within a larger network.

Distribtued caches can dramatically increase the available cache size. Whatever restrictions may exist on attaching cache storage space to an individual computer, distributed cache systems can get around them. Ignoring data replication issues for now, cache space should increase linearly with the number of computers in the cache. As is discussed in other parts of the chapter, this should logarithmically increase cache hit rate. Even without particularly intelligent distributed cache algorithms, we can realize a substantial gain.

Alongside increased storage space, a distributed cache throws more compute power at responding to user’s page requests. This allows a proxy cache system to handle more users at one time, and allows the increased storage space to be shared among more users. So we get more than just more processing power and cache space; many users can share a common cache space when large distributed caches are employed, potentially increasing performance as each cache’s overhead is amortized across more users.

Lastly, we have the issue of spreading the cache's entry points around over the WWW. The main goal here is latency reduction, so the latency incurred between the client and proxy cache should be less than that between the client and server. It it therefore advantageous for a proxy cache to have access points that are distributed around the WWW.

The issue of how to insure that clients use the closest access point it addressed later, but there is a simple corollary for single-system caches. The standard web proxy cache architecture - one machine that serves as a caching bridge between a private network and the WWW - is designed to take advantage of proximity. For this case there is a single entry point, so no effort to pair clients with the closest server is needed, but it highlights the importance of proximity in proxy cache design.

8.2.1 Handling Requests

Distribtuted web proxy caches introduce additional complexity over single-system caches because good load balancing within the cache is now essential for getting good performance. There are two basic classes of approach here: virtualizing access to the cache and routing requests internally.

Virtualizing cache access is the simpler technique to implement. The basic idea here it to provide more than one entry point to the cache, provide the appearence of a single access point, and then send each incoming request through to a real access point. Requests are received directly by each of the different access points and then processed.

This can be implemented using the same hostname mapping tricks that are frequently used to build large web servers. Providing a dynamic mapping between "www.myproxycache.com" and a set of caches "www[0...100].myproxycache.com" builds a very simply distributed cache. This system would even work if each cache operated independently of all the others. But using such a naive approach would decrease the potential performance of the system.

More performance can be achieved by being more careful about the load balance among the constituent systems. Load for a web proxy cache consists of three things: storage space for cached objects, downloading new web objects to cache, and replying to client requests. Ideally the load among these three tasks could be simultaneously balanced across all consitituent systems; this is probably too much to ask.

The simple dynamic hostname mapping system tried to balance the third factor, client requests. Balancing the other two tasks requires more communication within the cache.

8.3 System Architectures

After considering the load balance issues in a distributed cache design, the next step is to architect the structure of the cache. There are three basic communication structures for distributed caches:

- replicated
- peer-to-peer
- hierarchical

8.3.1 Replicated

The replicated cache is the simplest, and is exemplified by the dynamic mapping example above. In this case each system involved in the cache acts independently, with some mechanism for sending client requests to different cache systems. There are no internal

communication issues - becuase there is no internal communication. This architecture only addresses the client request load issue, but it can do an excellent job there.

8.3.2 Peer-To-Peer

One step beyond the replicted architecture is the peer-to-peer web proxy cache architecture. In these cases all of the constituent systems inter-communicate as equals in an attempt to share load equally. There are many different implementations of this type with a wide variety of goals.

8.3.3 Hierarchical

Even more abiitious and capable than the peer-to-peer systems discussed above are the hierarchical web proxy caches. In these systems the constituent caches communicate in an attempt to improve performance, but they are not all treated as equals. Generally there is a control ordering among caches, which some caches handling the processing of requests while other systems communicate with clients and servers.

Within this class of architectures, a common technique to improve performance is "URL Routing." These systems separate receiving requests from the processing of requests. Further, they partition the set of URL's that are handled by each request processor. When a URL enters the system it is passed to the constituent cache that it responsible for making caching decisions for it, and the response to the request originates there.

This techique is useful when a cache sees a lot of requests for a few domains and would like to be able to provide better caching performance for those requests based on the observation. For example, the users at one company may all frequently check CNN's web pages for news updates, ESPN's web pages for sports information, and then a large variety of other web pages on an individual basis. It makes sense of segregate proxy cache space and processing power for handling ESPN and CNN because we know that those pages will be accessed frequently. A proxy cache could use an internal URL routing system to insure that one part of the distributed cache was reserved for CNN, another for ESPN, and then continue building the rest of the cache as needed. This would insure that requests for CNN are treated on their own system and are not bogged down by a users request for a page that is not currently in cache.

8.4 Synchronization Issues

When building a distributed cache we face the same synchronization and data consistency issues within that cache that web proxy caches face within the larger context of the WWW. If a distributed cache stores the same web object in more than one place, which version does the user get? The more recent copy? The one that is faster to return? A random choice? There is no right answer here, but there are significant design and performance implications to choose when deciding on the answer to use for a particular implemention.

Ideally we would always return the most recent version of the object - or an older copy that is identical. Using the language of the Cache Consistency section of this chapter, this requires implementing a strong consistency model within the distributed cache. And as was remarked before, this can significantly decrease performance.

We would also like to minimize the time taken to respond to the request, which makes the "fastest copy to retrieve" answer attractive. Of course it takes some time to compute which copy can be returned fastest, which might leave us with a variant of the random copy

solution. This is certainly the easiest to implement, and nicely parallels the weak consistency model discussed for web proxy caches in general.

And as there are a variety of ways to improve on true weak consistency for overall cache coherence, there are ways to build slightly "more consistent" internal consistency models. Much as a proxy cache can check every response against the originating server to guarantee the cached data is not stale, a distributed cache system could check responses against either the originating server or the full set of constituent caches.

8.5 Data Flow Models

After considering the consistency issues within our distributed cache, we must consider how the cache will interact with clients vis a vis retrieving new content. There are two basic approaches to this problem:

- request-cache-response caching
- "reverse" caching

Each has different performance characteristics, and much like all of the other major architectural issues, should be considered when designing a proxy cache.

8.5.1 Request-Cache-Response

This is the simplest proxy cache design based on the traditional request-response nature of many client-server systems. We use the three-part name "request-cache-response" to indicate the exact nature of this design when used in a proxy cache. The client sees a request-response system, but embedded within the processing to generate the response the proxy may generate another request-response data transfer. This recursive request-response should be differentiated from normal request-response because the server being accessed is itself a client for the same sort of protocol.

8.5.2 Reverse Caching

Many commercial web proxy cache companies implement a kind of caching in which data flow is inverted relative to the traditional proxy cache design. In these cases the proxy cache holds cached objects for originating servers according to an external relationship between content provider and proxy cache company. Instead of caching pages on demand as they are requested by users, these caches contain whatever pages the content provider dictates are to be cached (i.e. whatever pages they pay for). This turns proxy caching on its head, but still performs a similar function. Such caches preform the same latency reduction functions as other web proxy caching systems, but only for the targeted set of pages.

8.6 Meta-Architecture

Most existing web proxy caches are a mixture of these ideas. Further, the caching architecture most users find when accessing the WWW involves several autonomous caches: browser, maybe several proxies and some server-side caching. For the most part, these caches are fully autonomous - and for exactly the same reasons we used before to defend decisions to relax consistency requirements within distributed caches. Keeping such a large number of caches consistent would be extremely difficult and would likely kill most of the benefit from caching in the first place.

So what sequence of caches do we find when accessing information on the WWW? For low traffic sites, the answer is generally that only the browser cache comes into play. If the user accessing the information is configured to use a local proxy cache, then a second level of caching comes in. When the web object being requested comes from a high traffic site, it is likely a more complex transaction. The user's request is first checked against the local browser cache. If that misses the local proxy cache is checked.

So far the path is identical for all pages. If the proxy cache misses, it then needs to load the page from the originating server. This is where high-traffic sites behave differently. The server is likely using a server-side cache, a reverse caching service, or both. The proxy's attempt to load the page will then propagate through the server's cache, and finally retrieve the content from the reverse web caching service. One experiment revealed the following chain of events to access a page on the CNN web site:

- check against browser cache, miss
- check against local proxy cache, miss
- download page from CNN, mainly pointers to Akamai
- download content from Akamai

This listing ignores the cache-update process that takes place as the content travels up each level of the caching hierarchy.

The potential for hybrid cache architectures is great. Even the simplest web proxy cache often plays the role of a single caching entity within a large, weakly consistent, distributed caching system. It might be the case that the only communication within this system takes place as HTTP requests driven by user page-loads. But the entire collection of systems involved in delievering content to the user can be thought of as the constitents of a large caching system. And all of the caching taking place between the browser cache and the server's internal caching are part of the large, distributed proxy cache seen by the user.

/

9 Performance Measurement

9.1 Introduction

As the World Wide Web growing, caching proxies become a critical component to reduce both network traffic and client latency. However, there has been little understanding the performance between different proxy servers, and their behavior under different workloads. So it is critical to design and implement a proxy benchmark to test and understand the performance characteristics of a proxy server. Using a benchmark, customers can not only test the performance of a proxy running on different software and hardware platforms, but also compare different proxy implementations and choose one that best matches the customer's requirements.

In this section, we mainly describe two well-known cache benchmarks: the Wisconsin Proxy Benchmark and Poly-graph.

9.2 The Wisconsin Proxy Benchmark

9.2.1 Introduction

The Wisconsin Proxy Benchmark (WPB) [2] was one of the first publicly available cache benchmarking tools. The main feature of WPB is that it tries to replicate the workload characteristics found in real-life Web proxy traces. WPB consists of Web client and Web server processes. First, it generate server responses whose sizes follow the heavy tailed Pareto distribution described in [12]. Since heavy-tail distribution of file sizes impact proxy behavior, it is important to include very large files with a non-negligible probability. As it must handle files with a wide range of sizes. Second, the benchmark generates a request stream that has the same temporal locality as those found in real proxy traces. Third, Since the benchmark is often run in a local area network and there is no nature way to incur long latencies when fetching documents from the servers, the benchmark let the server process delay sending back responses to the proxy to emulate Web server latency. However, Web server latencies affect the resource requirements at the proxy system. Thus, the benchmark supports configurable server latencies in testing proxy systems.

The main performance data collected by the benchmark are latency, proxy hit ratio, byte hit ratio, and number of client errors. There is no single performance number since different environments weight the four performance metrics differently. Proxy throughput is estimated to be the request rate dividing by the request latency.

9.2.2 Distinguishing Features of WPB

The distinguishing features of WPB include:

- Support for studying the effect of adding disk arms.
- The effect of handling lowbandwidth (modem-based) clients.

9.2.3 General setup of the benchmark

The General setup of the benchmark is that a colection of Web client machines are connected to the proxy system under testing, which is in turn connected to a collection of Web server machines. There can be more than one client or server processes running on a client or s server machine. The client and server processes run the client and server codes in the benchmark, instead of running a browser or a Web server. There is also a master process to coordinate the actions of client processes and generate an overall performance report. Some of the setup parameters are defined in a configuration file.

9.2.4 Performance of Example Systems

WPB is used to measure the performance of four proxy systems: Apache version 1.3b2, Cern version 3.0A, A Cern-derived comercial proxy and Squid version 1.1.14. A set of experiments using WPB have been run to analyse how the above systems perform under different loads. The number of client processes and collected statistics for caching and no caching configurations was different. The simulation results are shown as following:

- Although there are their vast differences in implementations, the performances of Cern and Squid are comparable. Squid mainly suffers from not being able to use the extra processor in the multi-processor system. Cern, on the other hand, uses a process-based structure and utilize two processors. In addition, CERN takes advantage of the file buffer cache, which seems to perform resonably well.

- In terms of latency, due to the two-phase store, that introduces extra overhead, Apache has the worst performance. Proxy N has a slightly better performance overall. However, this may be a consequence of the great number of errors. Since a smaller number of requests are effectively handled, that can reduce the delays for contention.

- In terms of hit ratios, Squid and Apache maintains roughly constant hit ratios across the load. For both Cern and proxy N, hit ratio decreases significantly as the number of client increases.

9.2.5 Effect of Adding Disk Arms

Using WPB, the impact of spreading the cached files over multiple disks on proxy performance has been analysed. The simulation results indicate that disk is the main bottleneck during the operation of busy proxies. Adding an extra disk reduces the bottleneck in the disk. However, for Squid, this reduction did not reflect in an improvement in the overall performance of the proxy. For proxy N, an improvement of 10% was achieved.

9.2.6 Effect of Low Bandwidth Client Connections

The impact of low bandwidth connections on proxy performance has been analysed. A modem emulator which introduces delays to each IP packet transmitted in order to achieve a certain effective bandwidth that is smaller than the one provided by the network connection. Simulation results show that: When a proxy must handle requests sent throught very low bandwidth connections, the time spent in the network dominates. Both disk and CPU remains idle for more than 70% of time. As a consequence, proxy throughtput decreases and client latency increases by more than a factor of two.

9.2.7 Conclusion

Some interesting findings through use of WPB are the following:

- By increasing the number of disks, queueing overheads are reduced, the time spent servicing each disk request are also shortened. To some proxy caching systems, this also reflects on the overall performance of the proxy.
- Latency advantages due to caching are essentially erased when considering the overall profit to modem-based clients.

While WPB addresses a number of important benchmarking requirements, such as initial support for temporal processes, it has some limitations. These include lack of support for modeling spatial locality, persistent HTTP 1.1 connections, DNS lookups, and realistic URLs.

9.3 Polygraph

9.3.1 Introduction

Polygraph [35, 7] is a recently developed, publicly available cache benchmarking tool deeloped by NLANR. It can simulate web clients and servers as well as generate workloads that try to mimic typical Web access. Polygraph can be configured to send HTTP requests through a proxy. High-performance simulation allows to stress test various proxy compoents. The benchmarking results can be used for tuning proxy performance, evaluation of caching solutions, and for many other interesting activities.

Polygraph has a client and a server component, each of which uses multiple threads to generate or process requests. This allows Polygraph to simulate concurrent requests from multiple clients to multiple servers. Polygraph can generate different types of workload to simulate various types of content popularity. For example, requests can be generated which obey a Zipf-like distribution, which is largely believed to be a good estimate of real web usage patterns[9].

Polygraph includes two programs: polyclt and polysrv. Poly-client(-server) emits a stream of HTTP requests with given properties. The requested resources are called objects. URLs generated by Poly-client are built around object identifiers or oids. In short, oids determine many properties of the corresponding responese, including response content length and cachability. These properties are usually perserved for a given object. For example, the response for an object with a given oid will have the same content length and cachablility status regardless of the number of earlier requests for that object.

As it runs, Polygraph collects and stores many statistics, including: response rate, response time and size histograms, achieved hit ratio, and number of transaction errors. Some measurements are aggregated at five second intervals, while others are aggregated over the duration of the whole phase.

9.3.2 Distinguishing Features of Polygraph

The distinguishing features of Polygraph include:

- It is capable of generating a whole spectrum of Web proxy workloads that either approximate real-world traffic patterns, or are designed to stress a particular proxy component.
- Polygraph is able to generate complex, high request rate workloads with negligible overhead.

9.3.3 Conclusion

Polygraph is a high performance cache benchmarking tool. It can evaluate the performance of various caching systems, using cache-specific performance metrics, such as amount of bandwidth saved, response time, hit rate, and various scalability metrics.

More recently, Polygraph has been playing an increasing role in holding open benchmark Web Caching Bake-off's as a way of inspiring the development community and encouraging competition towards good caching solutions. A summary of their study comparing a number of commercial and academic systems can be found at [37].

9.4 IRCache Web Cache performance bake-offs

9.4.1 Introduction

Bake-off [37, 38, 36] implies testing several independent implementations of similar products, taking place within a short period of time and usually at the same location. Every product is tested under the same conditions. Bake-off results are used to evaluate the performance of Web caching proxies.

9.4.2 Why Bake-offs

- Fair Competition

  Test labs, audited on-site tests, and even SPEC-like reports are considered to be "fair". That is, they give equal opportunities to participants to win. So what makes Caching bake-offs special?

  The primary reason is highly competitive and unstable environment. New product releases and even companies appear virtually every month. Benchmarking workload improvements are also quite common. In such atmosphere, two test results obtained a few month apart are by default unfair to compare. A vendor who gets the last test "slot", essentially has a big advantage over the vendor who opened the test sequence. Thus, we have to test a lot of products in a short time interval.

- Test auditing

  Web proxy benchmarkign often requires complex test setup that involves many sophisticated componets like client-server simulators, L4 switches, and clusters of caches. Our experience shows that expertise and a lot of extra effort is required to guarantee the correctness of the setup. The auditing requires physical presence of the auditor during all stages of the tests.

To summarize, fair competition objective implies semi-concurrent execution of tests while test auditing requires human monitoring of test execution. It is currently infeasible to provide high quality auditing for dozens of isolated companies within a short period of time. Thus, only bake-off format produces fair and high quality results.

9.4.3 Web Polygraph

Polygraph is a high-performance proxy benchmark. It can generate about 1000 requests per second between a client-server pair on a 100baseT network. Furthmore, Polygraph allows you to specify a number of important workload parameters such as hit ratio, cachability, response sizes, and server-side delays.

- The cache-off Workload: PolyMix-3

  The PolyMix environment has been modeling the Web traffic characteristics since PolyMix-2. PolyMix-3 combines the fill and measurement phases into a single workload. The benefit to this approach is that the device under test is more likely to have steady state conditions during the measurement phases. Also, a larger URL working set can now be formed without increasing the duration of a test. And PolyMix-3 servers use a Zipf(16) distribution to close active connections. The servers also timeout idle persistent connection after 15 sec of inactivity, just like many real servers would do.

  For detailed treatment of many PolyMix-3 features, please check the Polygraph Web site: http//polygraph.ircache.net/.

9.4.4 Benchmarking Environment

- Polygraph Machines

  250 PC's were used as Polygraph clients and servers. And FreeBSD-3.4 were used as the base operating system for the Polygraph clients and servers.

- Time Synchronization

  The xntpd time server was run on all Polygraph machines and the monitoring PCs. The monitoring PCs are synchronized periodically with a designated reference clock.

- Network Configurations

  Each test bench consists of Polygraph machines, the monitoring PC, the participants proxy cache, and a network to tie them together. The routed network configuration uses two subnets. The clients, proxies, and monitoring PC use one subnet, while servers use the other. Bidirectional netperf tests were run between each client-server pair to measure the raw TCP throughput.

- Test Sequence

  This section describes the official testing sequence. The complete sequence was executed at least once against all cache-off entries. It inlcudes PolyMix-3, Downtime Test and MSL Test.

9.4.5 Comparison of Results

Please check the web site for detail information:http//polygraph.ircache.net/.

9.4.6 Conclusion

Using a single tool to benchmark multiple products does not necessarily allow for legitimate comparison of results. As we all know, when comparing results, it is very important to minimize differences in the testing environment. A seemingly minor difference in configuration may cause a very significant change in performance.

In order to compare the performance of different caching products under identical conditions, we proposed to hold a "bake-off". The basic idea is that everyone comes together for a few days in one location and puts their products through a series of tests. Because all tests occur at the same time and place, under identical conditions, comparisons ca be made between the participant's results.

In the chapter above, the IRCache Web Cache Bake-off was introduced. Simulation results show that the Bake-off is a high quality, independent verification of product performance in the Web caching community.

Bibliography

[1] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox, *Caching proxies: limitations and potentials*, Proceedings of the 4th International WWW Conference (Boston, MA), December 1995, `http://www.w3.org/pub/Conferences/WWW4/Papers/155/`.

[2] Jussara Almeida and Pei Cao, *Measuring proxy performance with the Wisconsin proxy benchmark*, Computer Networks And ISDN Systems **30** (1998), no. 22-23, 2179–2192, `http://www.elsevier.nl/cas/tree/store/comnet/sub/1998/30/22-23/2053.pdf`.

[3] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira, *Characterizing reference locality in the WWW*, Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS) (Miami Beach, FL), December 1996, `http://cs-www.bu.edu/faculty/best/res/papers/pdis96.ps`.

[4] Martin Arlitt, Ludmilla Cherkasova, John Dilley, Rich Friedrich, and Tai Jin, *Evaluating content management techniques for Web proxy caches*, Proceedings of the Workshop on Internet Server Performance (WISP99), may 1999, `http://www.cc.gatech.edu/fac/Ellen.Zegura/wisp99/papers/arlitt.ps`.

[5] Martin F. Arlitt and Carey L. Williamson, *Web server workload characteristics: The search for invariants*, Proceedings of ACM SIGMETRICS, May 1996.

[6] Michael Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm, *World-Wide Web caching – the application level view of the Internet*, IEEE Communications Magazine **35** (1997), no. 6.

[7] G. Barish and K. Obraczka, *World wide web caching: Trends and techniques*, 2000.

[8] Tim Berners-Lee, *Hypertext transfer protocol – HTTP/1.0*, HTTP Working Group Internet Draft, October 1995.

[9] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, *Web caching and Zipf-like distributions: Evidence and implications*, Proceedings of the INFOCOM '99 conference, March 1999, `http://www.cs.wisc.edu/~cao/papers/zipf-like.ps.gz`.

[10] Pei Cao and Sandy Irani, *Cost-aware WWW proxy caching algorithms*, Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97) (Monterey, CA), December 1997, `http://www.cs.wisc.edu/~cao/papers/gd-size.ps.Z`.

[11] Vincent Cate, *Alex – a global file system*, Proceedings of the USENIX File System Workshop (Ann Arbor, Michigan), May 1992, `http://ankara.bcc.bilkent.edu.tr/prv/ftp/INFO/Internet/Alex/usenix.wofs92.ps`, pp. 1–11.

[12] Mark Crovella and Azer Bestavros, *Self-similarity in World-Wide Web traffic evidence and possible causes*, Proceedings of the SIGMETRICS '96 conference, May 1996, `http://cs-www.bu.edu/faculty/best/res/papers/sigmetrics96.ps`.

[13] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella, *Characteristics of WWW client-based traces*, Tech. Report BU-CS-95-010, Computer Science Department, Boston University, 111 Cummington St, Boston, MA 02215, July 1995, `http://www.cs.bu.edu/techreports/95-010-www-client-traces.ps.Z`.

[14] John Dilley and Martin Arlitt, *Improving proxy cache performance-analyzing three cache replacement policies*, Tech. Report HPL-1999-142, Internet Systems and Applications Laboratory, HP Laboratories Palo Alto, October 1999.

[15] Adam Dingle and Thomas Partl, *Web cache coherence*, Proceedings of the 5th International WWW Conference (Paris, France), May 1996, `http://www5conf.inria.fr/fich_html/papers/P2/Overview.html`.

[16] Bradley M. Duska, David Marwood, and Michael J. Freeley, *The measured access characteristics of World-Wide-Web client proxy caches*, Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97) (Monterey, CA), December 1997, `http://www.cs.ubc.ca/spider/marwood/Projects/SPA/wwwap.ps.gz`.

[17] Venkata Duvvuri, Prashant Shenoy, and Renu Tewari, *Adaptive leases: A strong consistency mechanism for the World Wide Web*, Proceedings of IEEE INFOCOM'2000, March 2000.

[18] Steven Glassman, *A caching relay for the World-Wide Web*, Proceedings of the 1st International WWW Conference (Geneva, Switzerland), May 1994, `http://www1.cern.ch/PapersWWW94/steveg.ps`.

[19] James Gwertzman and Margo Seltzer, *World-Wide Web cache consistency*, Proceedings of the 1996 Usenix Technical Conference (San Diego, CA), January 1996, `http://www.eecs.harvard.edu/~vino/web/usenix.196/caching.ps`.

[20] Shudong Jin and Azer Bestavros, *GreedyDual* Web caching algorithms: Exploiting the two sources of temporal locality in Web request streams*, Proceedings of the 5th International Web Caching and Content Delivery Workshop, May 2000, `http://www.terena.nl/conf/wcw/Proceedings/S2/S2-2.pdf`.

[21] ———, *Popularity-aware greedydual-size algorithms for Web access*, Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS), apr 2000, `http://cs-people.bu.edu/jins/Research/popularity-aware.html`.

[22] ———, *Sources and characteristics of web temporal locality*, Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000.

[23] ———, *Temporal locality in web request streams : sources, characteristics, and caching implications*, Proceedings of the international conference on Measurements and modeling of computer systems, June 2000.

[24] Terence P. Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey K. MacKie-Mason, *Biased replacement policies for Web caches: Differential quality-of-service and aggregate user value*, Proceedings of the 4th International Web Caching Workshop, April 1999, `http://www.ircache.net/Cache/Workshop99/Papers/kelly-final.ps.gz`.

[25] Balachander Krishnamurthy and Craig E. Willis, *Piggyback cache validation for proxy caches in the World-Wide Web*, Proceedings of the 1997 NLANR Web Cache Workshop, June 1997, `http://ircache.nlanr.net/Cache/Workshop97/Papers/Wills/wills.html`.

[26] Balachander Krishnamurthy and Craig Wills, *Proxy cache coherency and replacement – towards a more complete picture*, Proceedings of the ICDCS conference, June 1999, `http://www.research.att.com/~bala/papers/ccrcp.ps.gz`.

[27] Balachander Krishnamurthy and Craig E. Wills, *Study of piggyback cache validation for proxy caches in the World Wide Web*, Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97), December 1997, `http://www.cs.wpi.edu/~cew/papers/usits97.ps.gz`.

[28] ———, *Piggyback server invalidation for proxy cache coherency*, Proceedings of the 7th International WWW Conference (Brisbane, Australia), April 1998, `http://www7.conf.au/programme/fullpapers/1844/com1844.htm`.

[29] Donghee Lee, Jongmoo Choi, Sam H. Noh, Sang Lyul Min, Yoohun Cho, and Chong Sang Kim, *On the existence of a spectrum of policies that subsumes the lru and lfu policies*, Proceeding of the 1999 ACM SIGMETRICS Conference, May 1999.

[30] Chengjie Liu and Pei Cao, *Strong cache consistency for the World-Wide Web*, Proceedings of the Works In Progress session of the OSDI '96 conference, October 1996, `http://www-sor.inria.fr/mirrors/osdi96/wipabstracts/liu.ps`.

[31] P. Lorenzetti, L. Rizzo, and L. Vicisano, *Replacement policies for a proxy cache*, Tech. Report LR-960731, Univ. di Pisa.

[32] Ari Luotonen and Kevin Altis, *World-Wide Web proxies*, Proceedings of the 1st International WWW Conference (Geneva, Switzerland), May 1994, `http://www1.cern.ch/PapersWWW94/luotonen.ps`.

[33] Norifumi Nishikawa, Takafumi Hosokawa, Yasuhide Mori, Kenichi Yoshida, and Hiroshi Tsuji, *Memory-based architecture for distributed WWW caching proxy*, Proceedings of the 7th International WWW Conference (Brisbane, Australia), April 1998, `http://www7.conf.au/programme/fullpapers/1928/com1928.htm`.

[34] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum, *The lru-k page replacement algorithm for database disk buffering*, Proceedings of ACM SIGMOD, 1993.

[35] A. Rousskov, *Polygraph. http://polygraph.ircache.net/.*

[36] Alex Rousskov and Duane Wessels, *The third IRCache Web cache bake-off – the official report.*, October 2000, `http://bakeoff.ircache.net/bakeoff-01/`.

[37] Alex Rousskov, Duane Wessels, and Glenn Chisholm, *The first IRCache Web cache bake-off – the official report.*, April 1999, `http://bakeoff.ircache.net/bakeoff-01/`.

[38] ———, *The second IRCache Web cache bake-off – the official report.*, January 2000, `http://bakeoff.ircache.net/bakeoff-01/`.

[39] Peter Scheuermann, Junho Shim, and Radek Vingralek, *A case for delay-conscious caching of Web documents*, Proceedings of the 6th International WWW Conference (Santa Clara), April 1997, `http://www.scope.gmd.de/info/www6/technical/paper020/paper20.html`.

[40] Jia Wang, *A survey of Web caching schemes for the Internet*, ACM Computer Communication Review **25** (1999), no. 9, 36–46, `http://www.cs.cornell.edu/Info/People/jiawang/web-survey.ps`.

[41] Roland P. Wooster and Marc Abrams, *Proxy caching that estimate page load delays*, Proceedings of the 6rd International WWW Conference, April 1997, `http://www.scope.gmd.de/info/www6/technical/paper250/paper250.html`.

[42] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, *Using leases to support server-driven consistency in large-scale systems*, Proceedings of the 18th International Conference on Distributed Computing System (ICDCS '98), May 1998, `http://www.cs.utexas.EDU/users/dahlin/papers/icdcs98.ps`.

[43] Neal Yong, *On-line caching as cache size varies*, Tech. report, Computer Science Department, Princeton University.

[44] George Kingsley Zipf, *Human behavior and the principles of least-effort*, Addison-Wesley Press, 1949.