Cache is better than Check

Ranjita Bhagwan, Kanishka Lahiri, Kirk Miller, Pratik Mukhopadhyay

1 Introduction

The explosive growth of the World Wide Web over the last few years has motivated extensive research in the area of web caching. The problem is created by inefficient utilization of a scarce resource — network bandwidth. This is manifested as large response times seen by a user running a web browser. To make matters worse, today's World Wide Web can hardly be regarded as having progressed anywhere beyond infancy. The need for stricter and more efficient utilization of network resources will grow with the increasing usage of Internet services. In this survey we examine recent efforts in web caching — the most commonly applied technique to cope with growing demand on the Internet.

The basic aim of web caching is to try and take advantage of the way users access the Internet to prevent re-transmission of documents. User access patterns for the World Wide Web have been found to exhibit three kinds of reference localities: temporal, spatial and geographical [8].

We examine how existing and proposed methods that try to exploit one or more of these characteristics achieve the following goals:

- Reduction of user perceived latency — objects should be cached in such a way that response time to user requests is minimized;
- Reduction of network bandwidth consumption — overall number of bytes per second delivered by a server across large distances on the Internet should be minimized;
- Load balancing — The caching scheme should attempt to prevent a particular server from being drowned by many simultaneous requests.

In this survey of Web Caching techniques, we use the following terminology: *Internet objects* refer to files, documents, multimedia objects, etc.; *clients* refers to everyone who is accessing the Internet; *servers* are locations where the original most up-to-date version of an object exists; *proxies* are all locations in between the client and the server which work as sites for caching objects.

The remainder of this study is organized as follows. In Section 2 we discuss how Internet traffic characteristics motivate a rethinking of traditional caching methods. In Section 3 we survey existing client-based schemes using proxy caches and associated protocols. Proxy cache replacement policies are discussed in Section 4. In Section 5 we look at current research in server-based caching. Caching forces objects to be replicated. Replicated objects require policies to maintain consistency which we study in Section 6. Finally in Section 7 we draw our conclusions and suggest directions for future work.

2 Web Traffic Characterization

To understand the need for caching Internet objects and the policies that guide such efforts it is necessary to have a quantitative idea of the characteristics of client access patterns and web server loads. We discuss some results based on an analysis performed from server logs of five web servers [5]. The servers studied included a laboratory level web server at the University of Waterloo, a department level server at the University of Calgary, a campus wide server at the University of Saskatchewan, and NASA's Kennedy Space Center server.

- **Object Size:** Most of the objects accessed are in the range of 100-100000 bytes. Only 10% of the accesses exceeded 100000 bytes. This suits caching because most commonly the cache will not be swamped by oversize objects.
- **Concentration of references:** It has been shown in [5] that most of the accesses actually occur on a small set of objects. In these studies it was found that around 90% of all accesses were made to about 10% of the stored objects.
- **Temporal Locality:** The degree of temporal locality observed is lower than that observed in traditional file systems [5].
- **Geographical Locality:** Across all five data sets observed, 75% and more of the requests come from remote clients. Among the remote requests, 80% come from clients in North America and 10% from clients in Europe. However only 10% of the networks account for 75% of the remote requests.

Design issues for web caching schemes are different from those in traditional file system caching because of the following:

- The entire object must be cached: While the granularity of file system caching is at the disk block level, that of the World Wide Web is at the object level. This is because whenever an object is referenced, the entire object is transfered — not parts thereof, while file systems may well be interested in only a few blocks of a large file.
- Miss penalty is lower [10]: A page fault can be 10^5 times more expensive than a hit in main memory, thus RAM hit rate should be near perfect. However miss-to-hit cost ratios for web caches rarely exceed 10:1 therefore a 50% hit ratio may also provide acceptable performance.
- One-timers [4]: Many objects are accessed exactly once and therefore while traditional methods would cache such "one-timers", web caches would try not to.

3 Client Cache Organization

The organization of the cache at the client side can be described as a set of proxy caches connected to each other and cooperating to varying extents. In most types of proxy cache organizations the connections between the proxy caches form a tree-like structure. The proxy caches are connected by means of a high bandwidth LAN, and the root serves as the WAN gateway. When a client desires to find an object (URL), it contacts a leaf-level proxy cache. If the object is found it is returned to the client, else the request is forward to the proxy which is the parent of this cache. If a miss occurs at the root of the cache hierarchy, then the request is forwarded to the remote site. Most early web caches were organized in this way, with hierarchies of depth varying from 1 to 3. Performance studies in [10] have shown that more levels in the hierarchy hurt rather than help performance.

In a more general organization, the proxy caches are organized into cooperating groups, connected by a network of arbitrary topology. Each proxy queries the other cooperating proxies in the group for objects not found in its cache. Only if an object is not found in any of the caches in this group is a request forwarded over the WAN. The level of cooperation between the caches varies from system to system: it may be restricted to trying to trap the misses on the other caches, or may include cooperative replacement policies to try for a global optimum in the proxy cache group.

This shows that proxy cache organization requires communication among the proxy servers, and hence a mutually agreed upon communication protocol. This protocol needs to be extremely lightweight or the overheads of caching will eclipse possible benefits. This requirement rules out having HTTP as the communication protocol. The most commonly used protocol at present is ICP, the Internet Cache Protocol [27]. More advanced protocols like Hyper Text Caching Protocol (HTCP) [1], and Microsoft's Cache Array Routing Protocol (CARP) [26] have also been proposed.

3.1 Internet Cache Protocol

The process of searching for an object proceeds as follows: The client sends out a multicast query for an object to the proxy caches, and waits for their responses. If it fails to receive a response within the specified time, it sends the request to the server. When a proxy cache receives a request for an object, it sends a reply indicating the presence of the object, or if the object is small enough, it sends the object back along with the reply. If the client receives a positive response from multiple proxies, it uses the round trip time of the response to decide the proxy from which to fetch the object. The detailed ICP algorithm is shown in Figure 1.

3.2 The Harvest Structure

The Harvest Cache [10] was the first cache designed with the WWW as its target, and it implemented a simple hierarchical cache. Each cache in the hierarchy independently decided whether to fetch the object from the server or from its parent's cache. As the hierarchy deepens the root cache becomes responsible for more and more clients, and so the hierarchy needs to be shallow. The decision as to which objects are cacheable is made based on the URLs, *e.g.* cgi scripts are not cached and for security reasons neither are password protected pages. Although to uniquely identify an object we need both its URL and MIME type, the cache ignores the MIME types while reporting hits. To reduce the costs arising from failed proxies, each proxy implements a *negative caching* policy. Whenever a retrieval failure occurs, the site at which the failure occurred is not considered as a possibility for a parameterized period of time with default value of 5 minutes. The Harvest cache system evolved into the Squid web cache [27], which is currently the most widely used proxy cache. The performance of Squid in terms of varying latency and hit ratio with varying number of clients is shown in Figure 2 and Figure 3. The results were obtained using the Wisconsin Proxy Benchmark[3]. The experimental setup used consisted of 40 Sun Sparc20s, each with two 66 MHz CPUs, 64 MB main memory and two 1 GB disks. The workstations were connected using 100 Mbps Ethernet links. The number of clients was varied from 8 to 96. The experiments were conducted in both caching and no caching configurations, each in two phases.

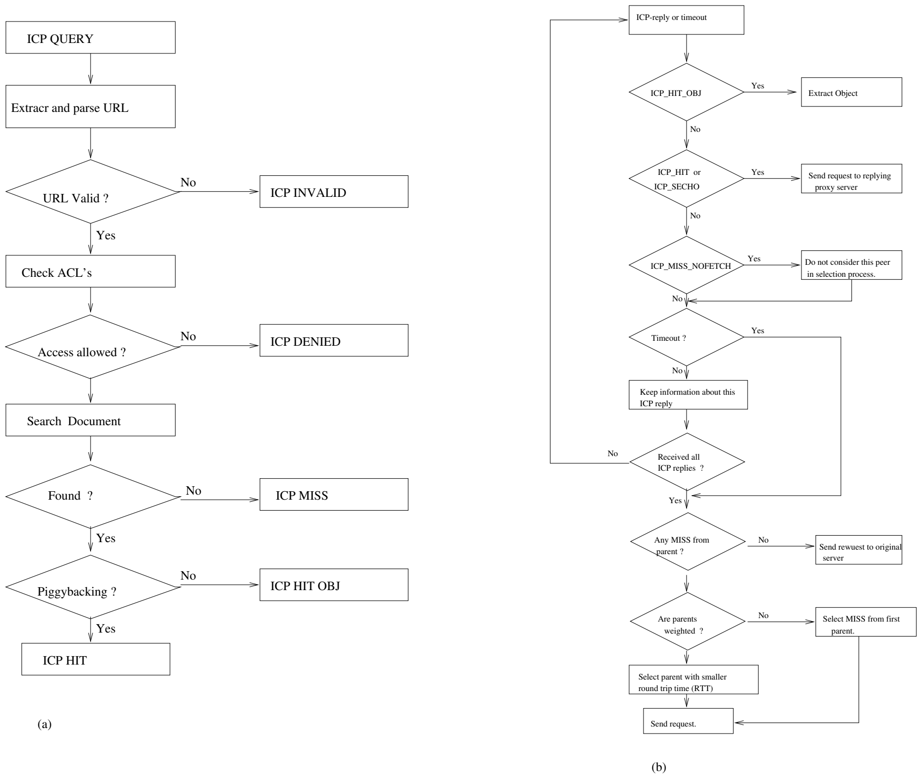

Figure 1: ICP actions at (a)responding and (b)requesting proxies[1]

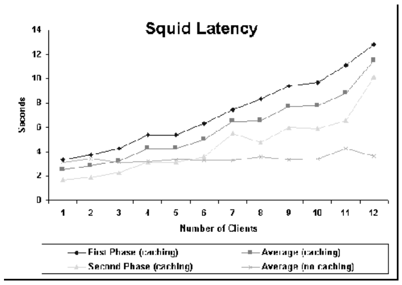

Figure 2: Latency for Squid 1.1.14 [3]

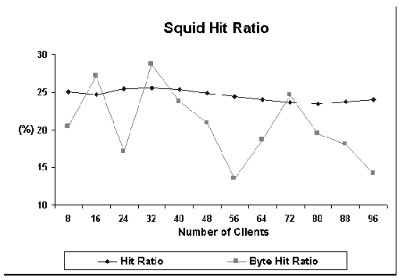

Figure 3: Hit Ratio for Squid 1.1.14 [3]

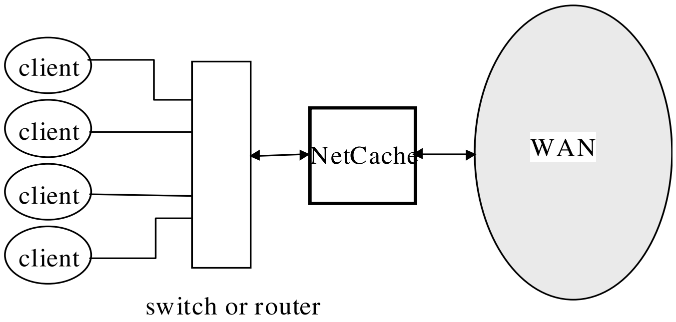

Figure 4: Organization of NetCache

3.3 Problems with ICP

ICP does not include "freshness" parameters (*e.g.* the *If-Modified-Since* header of HTTP, or *cache-control* header in HTTP 1.1) it is not possible to use "freshness" criteria in the cache resolution process. So if the different proxy caches have different "freshness" requirements it is possible for a client to receive data which is unsuitable (*i.e.* too stale) for it.

Also ICP has been found to scale poorly. If the process of cache hit resolution involves sending more than 4-5 ICP messages, the probability of a timeout occurring greatly increases, and this adversely affects the performance.

3.4 LSAM

Another design for a proxy cache is the LSAM proxy cache, proposed in [25]. This design tries to take advantage of the fact that some groups of pages become hot spots in the web. In the LSAM design, when a server detects that some of the objects it serves has become a hot spot, it starts a multicast channel broadcasting the relevant objects. When a proxy has a sufficient number of clients interested in a particular set of pages, the proxy subscribes to the multicast channel broadcasting those objects. This process is dynamic, based on the current load on the server and the proxy, and prevents any proxy/server from becoming overloaded.

3.5 NetCache

NetCache [11, 12] is the commercially marketed version of Harvest. It provides transparent caching using a cache cluster which many ISPs desire since it allows them to refurbish their networks with caching without making any changes at the client level.

3.5.1 Organization of NetCache

Transparency is achieved by using a NetCache proxy as the default gateway to a network. A simple situation is shown in Figure 4. Mapping of a client's TCP session to a cluster of web caches is done in three steps. First, when a client browser requests a particular URL, it sends a TCP SYN segment towards the WAN, with the purpose of setting up a connection with the server. The NetCache system, operating unknown to the client, intercepts the TCP SYN segment sent be the client and returns an acknowledgment, which causes the client to believe that it has established connection with the concerned web server. In the second step, NetCache selects a web cache from a set of caches by computing a hash function on the destination address. It then creates a TCP session between itself and the selected cache. Third, it acts as an intermediary between the client and the cache, forwarding requests from one to the other while giving the client the impression that it is connected directly to the web server.

Once the mapping to a web cache is accomplished, the NetCache system parses the client's HTTP request to reconstruct the URL of the requested object so that in case of a server failure, different IP addresses having the same URL can be tried to obtain the object.

The mapping of the HTTP requests of the client to the different web caches is done using a layer 4 (L4) switch [12]. It identifies TCP traffic that is headed for port 80 (the standard HTTP port) and diverts all such traffic to the web caches, while allowing other TCP packets to proceed into the WAN.

NetCache supports persistent TCP connections, both at the client and the server side. The client need not set up a fresh TCP connection each time it requests a URL from the same server. This reduces bandwidth usage and at the same time allows the client to access the requested object faster. Moreover the hash function used to map destination addresses to caches is designed to avoid replication of objects. This reduces network traffic needed to maintain consistency between different copies of the same object.

3.5.2 Routing Strategies

The cache clusters referred to above may be arranged hierarchically, with the requests made by the downstream cache cluster being mapped to the upstream cache cluster, in a manner similar to the way clients' requests are mapped to the local cache cluster. The structure is similar to that of Squid, but the advantage of Netcache over traditional ICP is that it avoids multicast queries and the network round trip time that is a characteristic of ICP.

An alternative to the hierarchical structuring is to maintain one large cluster of caches. A requesting client may be bound to any cache within the cluster. This cache then obtains the cacheable object from the cluster member to whom the requested URL hashes.

3.5.3 Weakness of Transparent Caches

Transparency tends to slow down the network connection because two TCP connections need to be established instead of one (one between the client and netcache system and one between the system and the actual web cache). Also, browsers drop persistent connections made as soon as they browse to a different web site. A more serious problem of transparent caching is that communication over the Internet for HTTP based objects is no longer end-to-end. In case a route suddenly becomes asymmetric, that is if the route from the client to the server no longer remains the same as that from the server to client, the network could appear partitioned for HTTP requests, while it wouldn't hamper other applications layer protocols at all.

3.6 Cache Array Routing Protocol (CARP)

There exist some inherent disadvantages of using a multicast-based protocol, as stated below.

- The use of multicast queries to determine if an object is cached or not can create extraneous traffic on an already congested network.
- The addition of more proxies to the hierarchy increases the number of such queries quadratically, which also increases the processing overhead [14]. In other words, distributed caching using ICP shows poor incremental scalability.
- ICP promotes replication of objects in the caches. This may cause large amounts of redundant objects to be stored in the cache hierarchy in the long run.

To overcome these problems, a new cache organization protocol known as Cache Array Routing Protocol (CARP) [26] has been proposed. This algorithm is based on maintaining a *array membership list* of *loosely coupled* proxies and a hash function which divides the entire URL space among these proxies. The membership list holds all necessary information regarding all the proxy servers in the array.

3.6.1 The CARP algorithm

The local proxy of a client computes a hash function for each of the member names in the array membership list. When a URL is requested, the local proxy server also computes a hash function for the requested URL and combines it with the array of hash functions computed from the membership list. The proxy for which the combined value is the largest is declared the *owner* of the specified URL. A request is now sent to the owner for the required object.

Both the URL and the member names are hashed so that when a specific member proxy fails, the overhead of re-routing a request can be reduced. For each member, a 32 bit value is calculated based on the characters in their names. While combining the two hash functions, CARP considers a *load factor* associated with each proxy. A proxy with greater processing power, larger memory, etc. is given a higher load factor so that a larger fraction of the URL space can be hashed onto it.

CARP provides a queryless deterministic caching mechanism as opposed to ICP. The partition of the URL space among the proxies is well defined and remains static until such a time that a new proxy is added to the array or an existing one is removed. There is always a fixed route from the requesting client to the proxy where the requested object is cached. CARP also scales well with the addition or removal of proxies. In such a situation, the algorithm recomputes the load factor for the proxies which changes the URL distribution.

3.6.2 Routing Strategies

Two alternative routing strategies can be adopted with CARP — hierarchical and distributed. In the hierarchical model, we refer to a proxy that is closer to the client than to the server as a *downstream* proxy, while one that is closer to the server than to the client is referred to as an *upstream* proxy. All requests made to a downstream proxy are forwarded to an array of upstream proxies. While making a routing decision, the downstream proxy refers to the membership list image that it has of the upstream array of proxies it is in direct contact with Figure 5. In the distributed model, all requests made to a proxy are forwarded to proxies within the same array. The two schemes can also be combined together to give a hybrid scheme.

In spite of all the advantages of CARP, it might not be an appropriate solution for cache sharing in WANs, since the network bandwidth between proxies is low. It does not address consistency issues. CARP also does not consider the geographical distances between clients and proxies, and hence may not function well when the caches are placed far apart from each other [14].

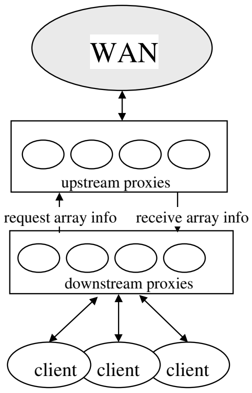

Figure 5: Hierarchical routing with CARP

3.7 CRISP

Caching and Replication for Internet Service Performance (CRISP) [16] has been proposed to cater to ISPs with large number of customers. It aims at integrating a number of proxy caches into one integrated shared cache. This is done using a *central mapping directory service* which contains caching information regarding which Internet object is cached in which proxy. The map is maintained either at one server or a number of servers, the latter providing more robustness to the system.

3.7.1 How CRISP works

When a client makes a request, it is routed first to the centralized mapping server through the client's local proxy. If the required object already resides in another proxy's cache, the map shows a *hit*. Information regarding the proxy where the object is stored is obtained from the mapping directory. The said proxy is then contacted by the local proxy and the object is retrieved from it. It is then forwarded to the requesting client. On the other hand if the map shows a *miss*, the requesting proxy fetches the object directly from the web server.

CRISP can be used to avoid multicast querying of ICP and at the same time maintain a comparable amount of robustness in the system. The path from client to cache is not deterministic as in CARP or NetCache and this makes it more resistant to failures of proxies. A central map makes consistency maintenance simple by allowing a modified site's web server to access it and invalidate the entries corresponding to the modified objects. Scalability of the CRISP system is also good because when a proxy joins the CRISP cache, it only needs to register itself with the mapping service. When a proxy fails, its inability to respond is detected by peer caches or by the mapping server. In such a case, the entries in the map corresponding to the failed cache are invalidated.

The CRISP system benefits from the use of a single, shared cache. Simulation studies have shown that sharing increases the number of cache hits significantly. Table 1 shows the average latency of CRISP's mapping server with six proxies and varying request arrival rates. The experimental setup used to obtain the results consisted of 500 266 MHz AlphaStation workstations linked by a switched 100 Mbps Ethernet. The first column shows an estimate of the smallest number of clients that can produce the query rate shown in the third column, while the second column gives the average number of clients. The average latency values are in the order of milliseconds, showing that the mapping server response time is not a bottleneck in CRISP's caching structure.

clients (conservative)	clients (average)	query rate (requests/second)	average latency
1605	84935	321	2.15 ms
1480	78320	296	1.79ms
895	47363	179	1.46 ms
470	247872	94	1.45 ms

Table 1: Mapping server query latency

3.7.2 Disadvantages of a Centralized Service

The fact that the entire mapping service is stored in only one place is a drawback, since failure of the mapping server would cause the whole shared cache to fail. This can be improved by statically partitioning the URL space among many different servers. However, failure of any one of them could cause serious problems to the clients. Hence the centralized nature of the scheme remains a problem of the CRISP caching structure. Another problem with CRISP is the round-trip latency between the proxies and the mapping service. This should typically be low. The constraint of keeping latency as small as possible limits the geographical expanse of the shared cache.

3.8 Crispy Squid

Proponents of CRISP argue that they can maintain the hit ratios obtained by Squid and other ICP caches and at the same time improve scalability and latency by using *Crispy Squid* [15], an extension of the existing Squid release. Three directory configurations have been prototyped with Crispy Squid. The first, *Partitioned Synchronous Directory*, divides the URL space among different mapping servers using hash functions in a way similar to CARP. In the second, *Replicated Asynchronous Directory*, the global mapping directory is replicated asynchronously so that copies are placed closer to the caches. In the third scheme, *Replicated Partial Directory*, only parts of the mapping directory are replicated asynchronously. Which parts to replicate can be decided in many ways, one of them being replication of only *shared objects*.

Further work on CRISP has shown that the aim of a web caching system should not be to blindly increase global hit ratios. When the caches are located far away from each other, it might be faster to access the web server to service a request instead of obtaining a copy of it from a remote cache. This is a possible scenario where various ISPs may be cooperating to share their proxy caches. The ISPs may form a well distributed network of caches, where geographical distances have to be taken into account while designing protocols for reducing network latency. A protocol has been suggested to take care of such situations [21]. It uses a *communication graph* with weights assigned to edges, which can be used to compute the path of minimum latency from the requesting client to the object. This is a new model for cooperative caching.

4 Replacement

In this section we briefly survey replacement strategies adopted at the proxy sites. There are numerous efforts at comparing performances of existing strategies, so we chose to highlight a few noteworthy points. For details of performance comparisons, the reader may refer [4, 2, 22, 3].

Methods used to measure performance of cache replacement strategies are diverse: [23] uses a measure of the "delay savings ratio"; often average latency measurements have been used as in [3], while [2] measures performance in terms of cost savings and robustness. However since cached Internet objects are of variable size, [6] argues that the performance metrics need to be measured in terms of *byte hit ratio* which is the ratio of total bytes hit to the total bytes requested, and *request hit ratio*, which is the ratio of number of hits to the number of requests to web objects.

The main issues dealt with in the surveyed material are as follows:

- The non-uniform nature of web object makes simple LRU a poor choice;
- A naive solution to incorporate the size of objects was SIZE — an algorithm where the largest object is removed first irrespective of last access time;
- Variations of LRU have been suggested to take into account the size of the object being cached:
  - LRUMIN [2] — this biases the LRU policy in favor of smaller objects. If the incoming object is of size S, the LRU object of size at least S is removed first. If there are none, the LRU object of size at least $S/2$ is removed and so on.
  - SIZE-MRU [22] — Objects above a certain threshold size are subject to MRU while the rest follow LRU.
- LFU [4] — Least Frequently Used objects are replaced. To take advantage of "one-timers" LFU*-aging is suggested [4]. LFU*-aging allows only objects with a reference count of 1 to be replaced. In the studies conducted in [4] this policy outperformed all others, generating about 90% hit ratios with a 16 MB cache. However an additional counter is required to decrement the reference count of an object over time.

More sophisticated methods have also been suggested. For instance, in [28, 23] the authors argue the most important factor in deciding to cache an object should be the difference in user latency times between the situations in which the object is cached and when it is not. In one algorithm[28], for each server the proxy maintains an estimated latency to open a connection to that server and the estimated bandwidth of the connection. Whenever a new document is received, the connection establishment latency and the bandwidth are measured and the estimates are revised. The replacement algorithm selects for replacement the document with the smallest estimated download time.

In theory many of the above schemes appear elegant and produce near optimal byte-hit-ratios. But it has been argued that many of these schemes (such as LFU-aging) are too expensive to implement [22].

5 Cache Consistency

As with traditional distributed file systems, when using caching on the WWW to increase performance, it is important to ensure that the cached object accurately reflects the data represented by the original. In many cases, the data from cached objects that can not be guaranteed to be consistent with the original is useless or unreliable. Caches must provide some level of consistency to allow objects to be useful.

5.1 Strong Consistency

The cache must provide a guarantee that the object provided is as up to date as the most current version on the original home site. This problem can be solved trivially by simply asking the home server for a new copy on every access to the object, but the whole point of caching is to avoid this. The methods for maintaining this consistency without going getting a new copy every time can be divided into two main categories: validation checks and invalidation signals.

5.1.1 Polling-Every-Time

Validation checks or polling-every-time provide a strongly consistent caching mechanism. Most servers and almost all file systems keep a time stamp of when an object was last changed. When a client requests an object it can check the validity of an object by sending the time stamp of copy of the object it has cached to the home server, and asking the server if it's version is up to date. If the cached version is invalid the server informs the cache and it supplies a fresh copy to the cache to give to the client. This guarantees that the newest version is always sent to the client, without the need to explicitly transfer the object very time.

This technique is actually supported well by HTTP. To get an object the client must send a HTTP GET message to the server or cache. When the header "If-Modified-Since:date" is appended to message then it become what is known as a conditional GET. If the object has not changed since the date provided in the header then a "Not Modified 304" reply is sent instead of the object. The response of the server to the conditional GET can be used to determine what to send to the client. If an object is returned it is forwarded to the client, otherwise the version in the cache is up to date and sent out.

The gains provided by this method are pretty simple to see. While the home server is still contacted upon every request, if the object is unchanged then the client is serviced by the cache and not the home site. If the object does not change very often then all the server has to handle are control requests for the object rather than numerous file transfers. The client still has to wait for the round-trip latency from the server and back. On the server side network traffic is significantly reduced if the object does not change very often, since control messages are quite small compared to most files.

5.1.2 Invalidation

Invalidation is another method that can be used to ensure consistency. A method used in many file systems, invalidation relies on the owner of an object to inform cached copies that they are out of date. This shifts responsibility of maintaining consistency from the caches to the server.

The server must maintain a list of the places where an object is cached. Whenever that object is changed the server sends out invalidation signals to all sites that have it.

The main drawbacks of this method are that there is no support for it and some believe [13] it may not scale well. There really is no protocol for sending out invalidations on the Internet. Even if there was, it would have to guarantee that all copies register themselves with the server, else they would not be informed of the changes. The WWW is orders of magnitude larger than any other file system and if the object is heavily used and cached at many sites it may become prohibitively expensive for the server to send invalidations to every site.

On the other hand, if the object does not change often, or at all, then very few invalidations are ever sent out. Many objects on the WWW are unchanging, written once and left alone until removed. Thus the server will never have to send out any invalidations, and as long as it's copy is not invalidated, a cache can conveniently serve the client without contacting the server.

5.2 Relaxed Consistency

Both the invalidation and the polling techniques have their drawbacks. It is hard to maintain correct consistency in caches over such a large network without performance or implementation problems, so there have been quite a few attempts to relax the consistency requirement to improve performance.

5.2.1 TTL

Most of these methods seem to make some use of a *Time-To-Live* (TTL) for objects. The idea is simple — when an object is entered into a cache it is assigned a TTL. Until that TTL is up, that object is considered valid in the cache, after that it is marked invalid and must be updated from the home server on the next request.

HTTP provides some support for this method. There is an *Expires*: header that can be included with an object. This header directly provides information on how long an object may be considered valid. Unfortunately very few objects include a *Expires*: header, as most objects at their creation do not know how long they will last. Some documents, though, do lend themselves to this kind of information (flight schedules, calendar events, etc.).

If the *Expires*: header is not provided then some attempt to guess the TTL must be made. Methods for determining the TTL vary wildly. Some use a static assignment (*e.g.* 1 hour), others more dynamic methods. By taking other information from the headers and using it one can make a more educated guess at a TTL.

5.2.2 Adaptive TTL (Alex)

Adaptive TTL, otherwise known as the Alex protocol[9], takes advantage the fact that future updates seem to correspond to past updates. If a document changes a lot in the past then it will likely change soon in the future, or if it never changed much before it probably won't change much soon. Thus these methods tend to use the age of an object (or some fraction thereof) as a TTL for an object. The greater the TTL the larger chance for stale (inconsistent) objects to be returned, but also greater performance advantages as a cache can service requests directly for longer periods of time. In Adaptive TTL possible inconsistency and performance can be directly traded off at the discretion of the cache.

Adaptive TTL has the advantages that while an object is valid, as in invalidations, no traffic passes between server and cache, and like polling, the server need not know what the caches are doing. In a sense it gets the best of both worlds — performance and convenience, and is regarded as the best performing weak consistency protocol [18, 19].

The drawbacks of weakly consistent methods are well documented [18, 19]. The most obvious is the possibility of invalid objects being returned to the client. This rate of error can vary a lot with the method for determining TTL and from object to object. While many clients are willing to tolerate quite a bit of staleness in an object [13], others are not, and there is no way for the client to inform the cache how much it is willing to tolerate. Caches must be careful in how they assign TTL to avoid intolerable staleness in an object.

6 Server-Based Caching

Central to server-based approaches is the idea that the server is in a unique position to make informed decisions regarding when and where to cache objects based on access histories which are not easily available at the client or proxy site. These approaches are proposed as complements rather than substitutes for client-based caching.

6.1 Push Caching

One of the earlier works on server-based caching was *speculation*. Server speculation was proposed in [7] to improve the quality of documents cached near the client. When the server receives a request for object D_i it sends back not only D_i but also all D_js that appears likely to be accessed in the near future. It involves maintaining conditional probabilities of accesses for every pair of objects in the server. However this method adds to network traffic in attempting to increase hits in the client cache, therefore does not meet the goals we described in Section 1. Therefore we examine subsequent research on *Push Caching* that avoids this problem.

6.1.1 Geographical Push Caching

The approach described in [17] proposes geographical push caching to makes a server take advantage of the geographical locality in client requests. This research recognizes the importance but limitations of client based caching. The myopic nature of client-based approaches motivates the need for shared caches, as already seen in Section 3. The case for Push Caching is made by the argument that the server can take an educated guess as to when and where an object should be cached. The quality of this guess depends on how accurately the server knows the network topology and has accumulated access histories. However knowing the network topology is not easy, so it is predicted using geographical distances which are easy to determine.

The architecture of such an approach consists of modified HTTP servers and a replication service. The modified HTTP server accepts objects that may potentially be replicated while the replication service provides it with information regarding potential sites for replication, free space available and current load on each of those potential "server proxies".

A server decides to replicate an object when the number of requests to it exceeds a dynamically determined threshold. The decision of where to cache the object is taken naively by examining all potential caches and choosing the one that provides maximum benefit in terms of bandwidth reduction. An object may be cached at geographically distant places every time there is a burst of locality from a region. Then each of these caches will behave as original servers and may spawn further caching levels.

6.1.2 Server Dissemination

Along the same lines [8] suggests *server dissemination* as an effective technique to complement client-based caching to achieve traffic reduction and load balancing. Like [17] they propose existence of several server proxies. A many-many mapping exists from a set of servers to a set of proxies, because each server may belong to several clusters and thereby map to several proxies. The choice of how to cluster the servers is based on network locality — distances measured by the actual number of hops. The effectiveness of this scheme depends on the quantity and popularity of the data "disseminated". In this study explicit schemes for choosing what to disseminate have not been described, the thesis being that a small (but judicious) selection can result in noticeable bandwidth savings. In Figure 6 we see that choosing 10% (5 MB) of data to be disseminated can lead to 40-50% savings in bandwidth for a particular server — in this case, http://www.cs.bu.edu. The study of load variation down the dissemination tree suggests that too many levels of dissemination may lead to non-uniform loads at the downstream server proxies. However the load seen at the upstream proxies are stable and substantially lower in the presence of the dissemination model.

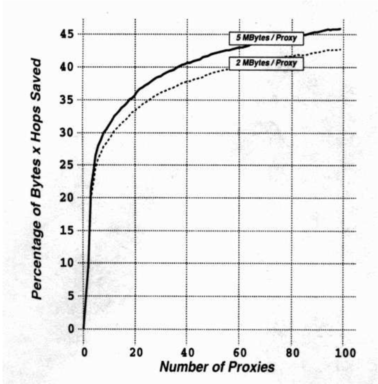

Figure 6: Bandwidth saved using server dissemination [8]

6.2 Main Memory Caching in Servers

Main memory caching is currently being re-examined in the context of server involvement in caching efforts. The problem is to efficiently cache objects in the main memory of servers to minimize the number of disk accesses and/or fetches though a network file system executed when serving requests. Note that this research is not aimed at reducing the consumption of network bandwidth but will benefit server load and user perceived latency. This is one of the reasons this problem has so far remained largely unaddressed in the context of web objects The first method we study dynamically adjusts to client's request patterns and cache size and the second statically determines the best set of objects to cache in main memory.

6.2.1 Adaptive Server Caching

This approach [20] is based on caching all objects that are below a certain threshold size. prevents very large rarely used objects from wastefully occupying the cache. Replacement is essentially LRU. The following are features that make this policy dynamic:

1. The value of the threshold is dynamically determined;
2. When a replacement is called for, the documents in the cache that are above the current threshold are replaced on an LRU basis. If there are none, or having done this even more space is required, the remaining objects are ejected on an LRU basis.

The value of the threshold is critical to performance. Through simulations it has been proved that a small threshold may lead to an under-utilization of the cache (most objects do not qualify to be cached) and a large threshold will cause thrashing (because many documents start competing for the same cache space) and in either case hit rates will fall. Also, there can be no "ideal" threshold, because the value that is optimal for one server at a given instance of time may not be optimal for another server or even the same server a few hours or days later. From trace driven simulations [20] it was observed that the optimal threshold for the NCSA server was 32 KB, a server whose objects average 17 KB. Similar studies for the Parallab (a supercomputing Center in the University of Bergen, Norway) show an optimal threshold of 4 KB when it was serving objects of average size 7.5 KB. In general the optimal threshold depends on the average size of objects delivered as well as the cache size.

The policy followed is as follows:

- start with an initial threshold;
- Periodically increase (decrease) the threshold. If the performance gets better, continue increasing (decreasing). Else start decreasing (increasing).

Comparisons are made between simple LRU caching, fixed threshold Caching and adaptive threshold Caching in Figure 6.2.1. In all cases - including the simple LRU scheme, the entire object is either cached or not cached. For fixed threshold caching hit ratios were determined for the "best" threshold and "worst" threshold. Trace driven simulations of five web servers show that the adaptive policy has hit ratios that are intermediate between the "best" and "worst" instances of fixed threshold policy. Interestingly, only one instance was found where the adaptive policy outperformed the "best" fixed threshold policy.

6.2.2 Static Server Caching

In [24] it is proposed that decisions need not be made dynamically because access patterns to web objects do not suddenly change. Thus it is worthwhile to consider "static" caching, where every once in a while the cache is refreshed

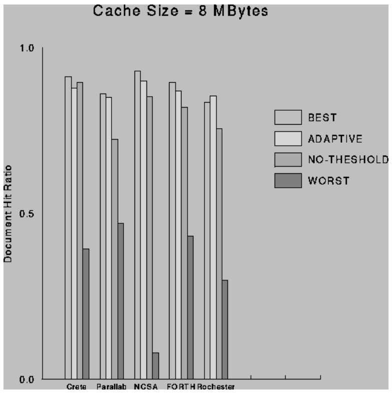

Figure 7: Performance of the adaptive policy [20]

with appropriate objects based on recent access characteristics. In between refreshes, the cache is not touched, but access data is collected. The advantage is that CPU overheads are lower, because no monitor process dynamically controls the value of a threshold hence balances server load.

The only parameter in a static caching policy is the period between refreshes. For example if the period is a day, once a day the server would access the log file of requests to analyze which were the most frequently accessed objects. Based on these frequencies it would choose an appropriate set of objects to cache for the following day.

The results of such a simple mechanism do not demonstrate an enormous improvement in hit ratios over other methods, but show consistent 10-12 % improvement for most of the cases studied. However if the data set size is small, as they observed in one of the analyzed servers, and traffic to that server is light the static caching algorithm will inefficiently utilize the relatively large cache space (for instance, it would "en-cache" all documents accessed on a given day). If many of these documents are not accessed on the next day, we will see a fall in performance.

The biggest advantage lies in the lower overhead suffered by the server in maintaining a set of cached objects. Refilling the cache is expensive (takes several minutes) but it is not performed frequently, and could possibly be performed during low server load periods. Also memory fragmentation caused by successive additions and removals from the cache is completely avoided - something all previous caching mechanisms suffer from. These leads to sub-optimal memory utilization or CPU overhead to overcome wastage. This seems a desirable feature especially in busy servers whose documents are rarely modified.

6.2.3 Server Architectures and More Replacement Strategies

There are efforts that directly address design of servers that are capable of handling high request loads. [22] describes the construction of a "server-cluster" consisting of a a set of "network nodes" and "storage nodes". Users connect to network nodes and secondary storage is attached to the storage nodes. [22] deals with replacement policies of caches maintained at both the network and storage nodes.

A scalable web server array architecture is described in [29]. The idea is similar to that in [22] but involves a central server that stores the entire document set and selectively broadcasts user requested documents to a surrounding array of "network nodes". The caching strategy adopted is based on a algorithm that preferentially caches objects with a high "byte access probability" defined as the ratio of the probability of an access of the object to its size. It is demonstrated in [29] that by caching 10% of the documents, server throughput achieved is the same as that obtained using mirrors, while the amount of disk space consumed is substantially less.

7 Conclusions

A lot of experimentation has been done by researchers to try to arrive at an optimal design for web caches. The main issues in web caching include organization, replacement policies and consistency maintenance. The performance of various replacement policies has been extensively compared by researchers, and their results show that simple strategies like LRU with size based admission control performs as well as more complicated admission strategies. While the hierarchical organization of proxy caches is the most widely used, other options like transparent caching are also becoming popular. Server-side caching and push based techniques are emerging as a possibility to remove server/network contention at the hot spots. While current web based applications do not require very strong consistency, this may change in the future, so consistency remains an important area of research. Future architectural developments (*e.g.* cheaper high bandwidth networks, smarter routers) will play an important role in determining the direction of research. Strategies for adaptive configuration of proxies based on current workloads is also an area for future work.

References

[1] M. Afonso, A. Santos, and V. Freitas. Qos in web caching. *Computer Networks and ISDN Systems*, 30(22-23):2093–2103, 1998. 3rd International Caching Workshop, Manchester, UK, 15-17 June 1998.

[2] Charu Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the world wide web. *IEEE transactions on Knowledge and Data Engineering*, 11(no.1):94–107, 1999.

[3] Jussara Almeida and Pei Cao. Measuring proxy performance with the wisconsin proxy benchmark. *Computer Networks and ISDN Systems*, 30(22-23):2179–2192, 1998. 3rd international Caching Workshop, Manchester, UK, 15-17 June 1998.

[4] Martin F. Arlitt and Carey L. Williamson. Trace-driven simulation of document caching strategies for internet web servers. *Simulation*, 68(1):23–33, 1997.

[5] M.F. Arlitt and C.L. Williamson. Internet web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–45, October 1997.

[6] A. Belloum and L.O. Hetrzberger. Document replacement policies dedicated to web caching. In *Proceedings of IEEE ISIC/CIRA/ISAS Joint Conference*, Gaithersburg, MD, USA, September 1998.

[7] A. Bestavros. Using speculation to reduce server load and service time on the www. In *Proceedings of the Fourth ACM International Conference on Information and Knowledge Management*, Baltimore, MD, USA, November 1995.

[8] A. Bestavros and Carlos Cunha. Server initiated document dissemination for the www. *IEEE Data Engineering Bulletin*, 19(3):3–11, September 1996.

[9] V. Cate. Alex-a global file system. In *Proceedings of the USENIX File System Workshop*, May 1992.

[10] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrel. A hierarchical internet object cache. In *Proceedings of the USENIX 1996 Annual Technical Conference*, 1996.

[11] Peter Danzig. Netcache architecture and deployment. *Computer Networks and ISDN Systems*, 30(22-23):2081–2091, 1998. 3rd international Caching Workshop, Manchester, UK, 15-17 June 1998.

[12] Peter Danzig and Karl L. Swartz. Transparent, scalable, fail-safe web caching. Technical Report TR-3033, Network Appliance, Inc.

[13] Adam Dingle and Thomas Partl. Web cache coherence. In *5th International World Wide Web Conference*, Paris, France, May 1996.

[14] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *Computer Communications Review*, 28(4):254–265, 1998. ACM SIGCOMM'98 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communication, Vancouver, BC, Canada.

[15] Syam Gadde, Jeff Chase, and Michael Rabinovich. A taste of crispy squid. In *Workshop on Internet Server Performance*, Madison, WI, USA, 1998.

[16] Syam Gadde, Michael Rabinovich, and Jeff Chase. Reduce, reuse, recycle: an approach to building large internet caches. In *Proceedings, The Sixth Workshop on Hot Topics in Operating Systems*, pages 93–98, Los Alamitos, CA, USA, 1997.

[17] J.S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, USA, May 1995.

[18] J. Gwertzmen and M. Seltzer. World wide web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, USA, Jan 1996.

[19] C. Liu and P. Cao. Maintaining strong cache consistency in the world wide web. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, May 1997.

[20] Evangelos P. Markatos. Main memory caching of web documents. *Computer Networks and ISDN Systems*, 28(7-11):893–905, May 1996. Fifth International World Wide Web Conference, May 6-10, 1996, Paris, France.

[21] Michael Rabinovich, Jeff Chase, and Syam Gadde. Not all hits are created equal: cooperative proxy caching over a wide-area network. *Computer Networks and ISDN Systems*, 30(22-23):2253–2259, 1998. 3rd international Caching Workshop, Manchester, UK, 15-17 June 1998.

[22] A.L Narasimha Reddy. Effectiveness of caching policies for a web server. In *Proceedings of Fourth International Conference on High-Performance Computing*, Bangalore, India, December 1997.

[23] Peter Scheuermann, Junho Shim, and Radek Vingralek. A case for delay conscious caching of web documents. *Computer Networks and ISDN Systems*, 29:997–1005, 1997. 6th International Conference on the World Wide Web, Santa Clara, CA, USA.

[24] I. Tatarinov, A. Rousskov, and V. Soloviev. Static caching in web servers. In *Proceedings of Sixth International Conference on Computer Communications and Networks*, Las Vegas, NV, USA, September 1997.

[25] Joe Touch and Amy Hughes. Lsam proxy cache: a multicast distributed virtual cache. *Computer Networks and ISDN Systems*, 30(22-23):2245–2252, 1998. 3rd International Caching Workshop, Manchester, UK, 15-17 June 1998.

[26] Vinod Valloppillil and Keith W. Ross. Cache Array Routing Protocol v1.0. Internet-Draft, June 1997.

[27] D. Wessels and K. Claffy. Icp and the squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–357, 1998.

[28] R.P. Wooster and M. Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29:977–986, 1997. 6th International Conference on the World Wide Web, Santa Clara, CA, USA.

[29] K.H. Yeung and K.W. Suen. Probability based replacement algorithm for www server arrays. In *Proceedings 1998 International Conference on Parallel and Distributed Systems*, Tainan, Taiwan, 1995.