Consistency Control Algorithms For Web Caching

Leon Cao

y2cao@math.uwaterloo.ca

February 9th, 2001

Abstract

This survey focuses mainly on the problem of cache consistency on the World-Wide Web (WWW). So far there is no perfect algorithm that deals with cache consistency issues that result from using web caching to reduce client latency and traffic. The survey first studies several web caching consistency approaches such as adaptive TTL, update threshold, polling-every-time, and invalidation, then compares various cache consistency algorithms used in transactional client/server DBMS architecture, trying to get a conclusion on what might be a good cache consistency algorithm for web caching. Each individual algorithm has its advantage given certain criteria. A good cache consistency algorithm should be able to deal with different user demands and network situations.

1. Introduction

The rapid increase in web usage has led to dramatically increased loads on the network infrastructure and on individual web servers. To ameliorate these mounting burdens, there has been much recent interest in web caching architectures and algorithms. Web caching reduces network load, server load, and the latency of responses. In the context of WWW, caches act as intermediate systems that intercept the end-users' requests before they arrive at the remote server. A web cache checks if the requested information is available in its local storage, if yes, a reply is sent back to the user with the requested data; otherwise the cache forwards the request on behalf of the user to either another cache or to the original server. When the cache receives back the data, it keeps a copy in its local storage and forwards back the results to the user. The copies kept in the cache are used for subsequent users' requests.

There are two basic types of web cache currently being used: *browser cache* and *proxy cache*. Browser cache is implemented by the local browser software such as Netscape or Internet Explorer. Basically the hard disk of the end user's computer has a section that stores objects the user accessed before. This cache is useful when the user hits the 'back' button to go to a page he/she visited before. Those cached pages will be read from the local cache almost instantaneously.

Proxy cache works on the same principle, but a much larger scale. Proxies serve hundreds or thousands of users in the same way; large corporations and ISP's often set them up on their firewalls. Because proxy caches usually have a large number of users behind them, they are very good at reducing latency and traffic. That's because popular objects are requested only once, and served to a large number of clients. Figure 1 is a general structure of web cache.

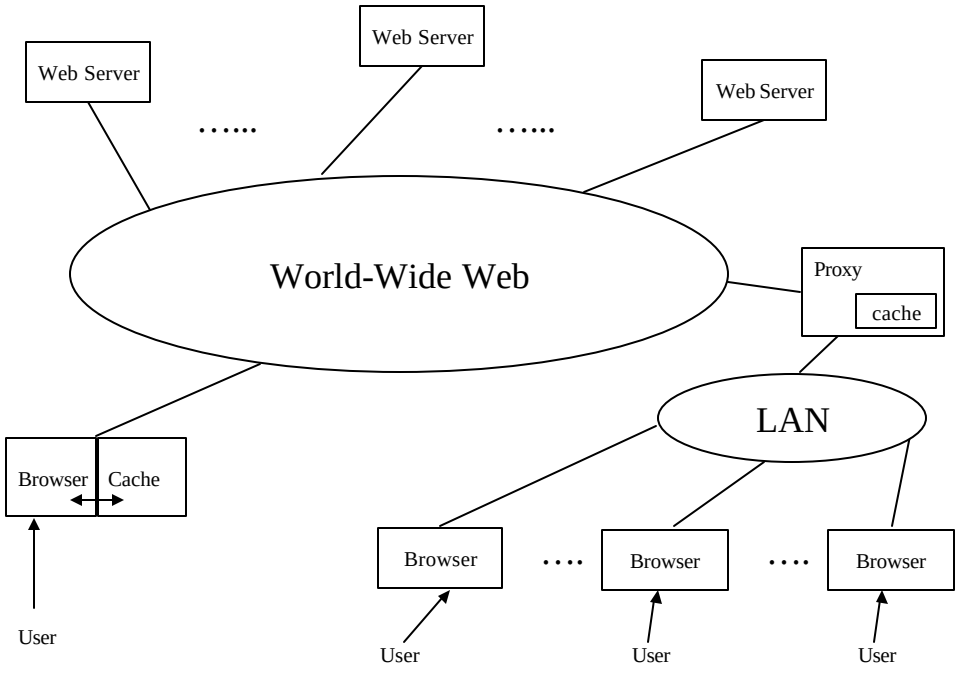

Figure 1: general structure of Web cache

However, caching has not only brought solutions on faster access to the Internet community, it has also introduced new issues and challenges. In essence, there is a pragmatic problem regarding the consistency between the cached data and the data on the

original server. By introducing caching mechanism, multiple copies of the requested data are created and stored in various caches all over the Internet. How to keep them consistent while considering the network traffic and client response delay, when to issue validation or invalidation message, which part, client cache or the web server, should issue such kind of message, these problems created an interesting area for research.

If we consider the heterogeneity nature of the WWW, the consistency problem becomes even more complicated. However, it is out of the scope of this survey. This survey will focus only on various approaches for current web caching, as well as existing cache consistency algorithms used in client-server architecture, assuming both client and server use compatible platforms.

The next section focuses on cache consistency approaches for the Internet; Section 3 gives comparison, limitations and possible improvements of these consistency approaches; Section 4 focuses on existing algorithms in client/server environment, their features based on a taxonomy. Section 5 concludes the survey and summarizes the possible future work to be done.

2. Cache Consistency for the Internet

The value of cache is greatly reduced if cached copies are not updated when the original data change. Cache consistency mechanisms ensure that cached copies of data are eventually updated to keep consistency with the original data. An ideal cache consistency solution will enforce the consistency to the maximum extent, while reducing the network bandwidth consumption and server load. There are basically two categories of cache consistency approaches: **weak cache consistency** and **strong cache consistency**. [3] gave a definition on these two terms. Weak cache consistency is the model in which client response time is more emphasized, but a stale document might be returned to the user; strong cache consistency, on the other hand, enforces the freshness of document all the time, but has the expense of extra server resource consumption.

2.1 Weak Cache Consistency

Existing Web caches mostly provide weak consistency, which means it is possible for the user to get a stale document from the cache, because at the moment, the contents on

the web server have changed but the cache hasn't done the synchronization yet. Two mechanisms fall into this category: *TTL (Time-To-Live)* and *Client Polling*. Their feature in common is that the client cache, who initiates the consistency process, sends validation messages to server.

2.1.1 TTL (Time-To-Live)

Under this approach, each object (document, image file, etc.) is assigned a TTL (time to live) value, such as two hours or one day. This value is an estimate of the object's lifetime, after which it's supposed to change. When the TTL expires, the data is considered invalid, and the next request for the object will cause the object to be requested from the original server. A slight improvement to this basic mechanism is that when a request for an expired object is sent to the cache, instead of requesting file transfer from the server, the cache first sends an "if-modified-since" control message to the server to check whether a file transfer is necessary.

TTL-based strategies are simple to implement, by using the "expires" header field in HTTP format. Following is an example of an HTTP header that applies the "expires" field:

HTTP/1.1 200 OK
Date: Fri, 09 Feb 2001 10:19:29 GMT
Server: Apache/1.3.3 (Unix)
Cache-Control: max-age=3600, must-revalidate
Expires: Fri, 09 Feb 2001 11:19:29 GMT
Etag: "3e86-410-3596fbbc"
Content-Length: 1040
Content-Type: text/html
…

The HTML document would follow these headers, separated by a blank line.

The challenge in supporting this approach lies in selecting an appropriate TTL value. If the value is too small, after a short period of time the cached copy will be considered stale. Therefore a number of "if-modified-since" messages will be sent to server frequently for expiration check, which results in extra network traffic (although it might be trivial comparing to the actual file transfer) and server overhead. On the other hand, if

the value is too big, out-of-date document may be returned to the end-user because the cache still thinks it's fresh while the document has already been modified by the server.

The critical issue for this approach is how to choose the TTL value. [2] initially used a flat lifetime assumption for their simulation, which means that they assigned all objects with equal TTL values. This resulted in poor performance in their experiment. Later on they modified the TTL value based on the popularity of the file. This is also mentioned in [3], which defined this improved approach as **adaptive TTL**. Adaptive TTL takes advantage of the fact that file lifetime distribution is not flat. If a file has not been modified for a long time, it tends to stay unchanged. [2] also mentioned that globally popular files are the least likely to change. By using adaptive TTL, the probability of stale documents is kept under reasonable bounds (<5%) [2].

2.1.2 Client Polling

Client polling is another approach of weak cache consistency. It means the client (cache) periodically checks back with the server to determine if cached objects are still valid. It is somewhat like the adaptive TTL, because under both cases the client sends out validation message to check if the document is still valid, when the client starts to think the document might be stale. Alex FTP cache [6] uses an **update threshold** to determine how frequent to poll the server. The update threshold is expressed as a percentage of the object's age. An object is invalidated when the time since last validation exceeds the update threshold times the object's age [2]. For example, consider a cached file whose age is 30 days and whose validity was checked one day ago. If the update threshold is set to 10%, then the object should be marked as invalid after 3 days (10% * 30 days). Since the object was checked yesterday, requests that occur during the next two days will be satisfied locally, and there will be no communication with the server including control message. After the two days have elapsed, the file will be marked invalid, and the next request for the file will cause the cache to retrieve a new copy of the file from the server. Same as TTL, the trick here is how to decide the update threshold.

2.2 Strong Cache Consistency

We could see from the introduction of weak cache consistency that weak consistency methods save network traffic and user latency at the expense of returning stale documents to the users. Under situations where document modification doesn't happen very frequently, or user doesn't have strict requirement on the freshness of the document, weak cache consistency is an economic approach. However, if the validity of the data is important (such as stock quote), then a strong consistency has to be enforced. The two widely accepted methods for strong cache consistency are *invalidation and polling-every-time.*

2.2.1 Invalidation

Many distributed file systems rely on this method to ensure that cached copies never become stale. In the web environment, this simple method also applies because conflicting updates never happen (if there are conflicting updates, what we need is a more complicated transactional consistency algorithm). Under this mechanism, the web server plays a crucial role. It is responsible for keeping track of the cached data. The server implements this by keeping all cache addresses that have the copy of the data. Once the data is modified on the server, the server sends out message to notify the caches on the list that their data are no longer valid. This notification process is considered complete only after all the clients on the list have received the message. Invalidation guarantees that when the user requests a document, the returned document is up-to-date. The trade-off is the overhead at server side.

2.2.2 Polling-every-time

This method is an extreme case of client polling that is used in weak cache consistency. Under this approach, whenever the client cache receives a document request from the end user and there happens to be a copy in the cache, it will first contact the web server to validate the cached copy. If it's fresh then the copy will be returned to the user, otherwise a new copy will be sent to cache and replace the old one. This approach also involves a lot of message transfer, possibly a large portion of which is useless. But given

a short lifetime of the objects and frequent requests from the user, this method is expected to be efficient.

3. Comparison, limitations and Improvements of Web Cache Consistency Approaches

3.1 Conceptual Differences

Let's take a look at the differences of the above mentioned cache consistency approaches from conceptual point of view:

- TTL, adaptive TTL and Client Polling, as the weak consistency methods, assign the central role to the client cache. Whether the end user gets a fresh document or not all depends on how often the cache checks the validity of the document. Therefore weak consistency methods are time-based, the document lifetime, or the update threshold determines the trade-off between document freshness and network bandwidth.
- Invalidation and Polling-every-time are event-driven. For invalidation method, the web server sends out invalidation message to caches only when its document is modified. For polling-every-time, whenever it receives a request from the end user, the client cache has to poll server to confirm whether the document is fresh. This also implies a possible heavy network traffic solely caused by all these control messages sent back and forth, which in turn results in the overhead of web server. The frequency of file update on the web server, or the frequency of user requests, greatly influence the network traffic and server load. Notice for strong cache consistency approaches, there is no trade-off to document freshness, because this is ensured by the protocol that the returned document is always fresh (if the time spent from server-send-out-invalidation-message to cache-receive-message could be ignored).

3.2 Experimental Results

Cao and Liu [3] implemented their experiments in a web caching system called Harvest, and compared the performance of adaptive TTL, invalidation and polling-every-time by replaying web server traces through the prototype running on workstations connected by an Ethernet. They used five Web server traces from the Internet Traffic Archive. Let's take the experiment results that got from two of them to analyze the limitations and benefits of each consistency approach. These two servers are: SASK (the Web server at the University of Saskatchewan, Saskatoon, Canada) and SDSC (the WWW server for the San Diego Supercomputer Center).

Trace	SASK, 51471 requests		
Modification	1148 files modified		
Approach	TTL	Polling	Invalidation
Hits	16456	16565	16268
Get Requests	35015	34906	35203
If-Modified-Since	922	16565	0
Reply 200	35388	35689	35203
Reply 304	549	15782	0
Invalidations	0	0	6028
Total Messages	71874	102942	76434
File Xfer bytes	185MB	187MB	183MB
Ctrl Msg bytes	3.91MB	7.09MB	4.29MB
Messages bytes	189MB	194MB	187MB
Stale Hits	< 410	0	0
Avg. Latency	0.124	0.138	0.134
Min Latency	0.010	0.039	0.010
Max Latency	32.1	12.2	107
Server CPU	26.0%	30.2%	27.6%
DISK RW/s	.37:2.2	.41:2.3	.41:2.5

Trace	SDSC, 25430 requests		
Modification	57 files modified		
Approach	TTL	Polling	Invalidation
Hits	4907	4907	4905
Get Requests	20523	20523	20525
If-Modified-Since	239	4907	0
Reply 200	20535	20549	20525
Reply 304	227	4881	0
Invalidations	0	0	248
Total Messages	41524	50860	41298
File Xfer bytes	263MB	263MB	263MB
Ctrl Msg bytes	2.39MB	3.38MB	2.36MB
Messages bytes	265MB	266MB	265MB
Stale Hits	< 14	0	0
Avg. Latency	0.16	0.173	0.165
Min Latency	0.010	0.038	0.010
Max Latency	12.2	12.2	12.2
Server CPU	34.1%	35.6%	32.7%
DISK RW/s	.94:2.3	1.4:2.0	1.0:2.2

Figure 2 Results from SASK & SDSC [3]

- The documents modified on SASK server is much more than those on SDSC. This is reflected by the number of different types of messages recorded on SASK and SDSC. Take TTL as an example, on SASK it created almost twice as GET requests as it did on SDSC. Compared to the modified-files ratio (1148/57), this is not too bad. The number of If-modified-since messages tripled, while again Reply 200 (reply with document follows) messages more than twice. One thing interesting and reasonable is that we could figure out the rough ratio of modified

files on each server by looking at the numbers of invalidation messages. 6028/248 = 24.3, while 1148/57 = 20.1, which is roughly the same.

- One result that worth paying attention to, I think, is the number of cache hits. Cache hits should be one of the most important factors to consider when evaluating the efficiency of a cache consistency algorithm, because it significantly reduces user latency. However, the numbers of cache hits in above two tables don't indicate the exact responsiveness and efficiency of the specific approach, because in the experiments, a cache hit is counted when the document that an end-user requests is in the cache. This doesn't necessarily mean that document is fresh. For invalidation, when a cache hit happens, the cached document will be sent to the user immediately; but for polling-every-time, when this happens, the client cache will send a 'if-modified-since' control message to server to confirm the freshness of the document. For adaptive TTL, this issue also exists. When a cache hit happens, the cache will first check if the document has expired according to the TTL value. If it's still valid the document will be sent out immediately, otherwise a control message will be sent to server.
- For each approach, the sizes of control messages on SASK are bigger than those on SDSC, which is due to the much bigger number of modified files. But SASK recorded a smaller file transfer size than SDSC. Probably this is because the files modified on SDSC have much bigger sizes.
- As invalidation messages are only generated in invalidation method, the stale hits only apply to weak consistency methods, i.e., adaptive TTL in [3]. If we compare the number of stale hits across the two tables, the one on SASK is much bigger than the number on SDSC, which result from the more frequency of file modification on the former.
- In both tables, we can see that contacting the server at each cache hit costs polling-every-time a much higher minimum latency and higher average latency than the other two approaches. The numbers also show that invalidation has a significantly large worst-case latency, i.e., a request from the end user can be stalled somewhere in the network for a long time. The authors of [3] pointed out

that this is because their current implementation could only handle incoming requests after all invalidation messages for a document have been sent via TCP.

- The last two rows in the above tables show the average server CPU utilization and disk read/write per second during the trace replay. Looking at the numbers, we could see that polling-every-time generally has a higher server CPU utilization, especially when the proxy cache hit ratio is high. This reflects the CPU cost of handling "if-modified-since" requests at the server.

The experiment results of [3] show that the invalidation approach performs the best among the three consistency approaches. Adaptive TTL works very well at keeping stale hit ratio low, but invalidation does not cost more in comparison. Whether it is in terms of cache hits, network traffic, response time, or server loads, invalidation performs quite similar to adaptive TTL, while it always ensures the freshness of the document. From this aspect invalidation approach is much more attractive than other cache consistency methods. Of course, invalidation is a preferred method for maintaining strong consistency than polling-every-time. The reason is straightforward. Except in the extreme case of file lifetime on the order of minutes, polling-every-time produces too many useless control messages sent to server for document validity check, which results in extra network bandwidth consumption and server load.

Gwertzman and Seltzer (G&S) [2] also used Harvest system to conduct their experiments. The difference is that they added Alex protocol [8] to their experiment, while eliminating the hierarchy factor of the Harvest system. This is because hierarchical caching structure significantly reduces the overhead for invalidation, which might not be a good factor in the experiments to compare these methods. The effect of hierarchical factor is also pointed out in [3], where the authors attributed their reason of eliminating the caching hierarchy to the fact that hierarchical caches are not yet widely present in the Internet.

G&S's simulation model is somewhat optimized by taking advantage of the combination of adaptive TTL and invalidation mechanisms. They used a lightweight cache server, which has an independent process that checks the freshness of the

documents periodically. Whether a document is stale or not is determined by using TTL values and invalidation callbacks from cooperating primary servers. The authors further optimized the invalidation protocol such that when the cache receives an invalidation notice, the document is marked as invalid but the cache will not retrieve the document immediately from server until the next user request comes. Although this will certainly increase latency in subsequent accesses to the document, it decreases bandwidth consumption if the document is not accessed again. The experiment results show that the cache miss rates improved dramatically because of this change.

After a couple of experiments, G&S realized that they should change the flat lifetime distribution of documents because basically, different types of documents have various lifetime as well as access frequency. They gathered information from Microsoft proxy server and Boston University's server log. Following is the data they summarized:

File type	%-age of total access	Average File size(B)	Average life-span (days)	Median age (days)
GIF	55%	7791	85	146
HTML	22%	4786	50	146
JPG	10%	21608	100	72
CGI	9%	5980	NA	NA
Other	4%	NA	NA	NA

From the table we could see that image files (GIF and JPG) have a relatively long lifetime which means they are less likely to change. Meanwhile their sizes are relatively small (in the order of KB). This indicates that they are good candidates for caching. Based on this the authors concluded that weak cache consistency will be effective since the most popular web objects also have the longest life-span.

G&S's experiments show that the update threshold approach provides the best performance among TTL, invalidation and itself. It could produce a stale rate of less than 5%. Meanwhile, it produces server load comparable to, or even less than that of the invalidation protocol with much less bookkeeping.

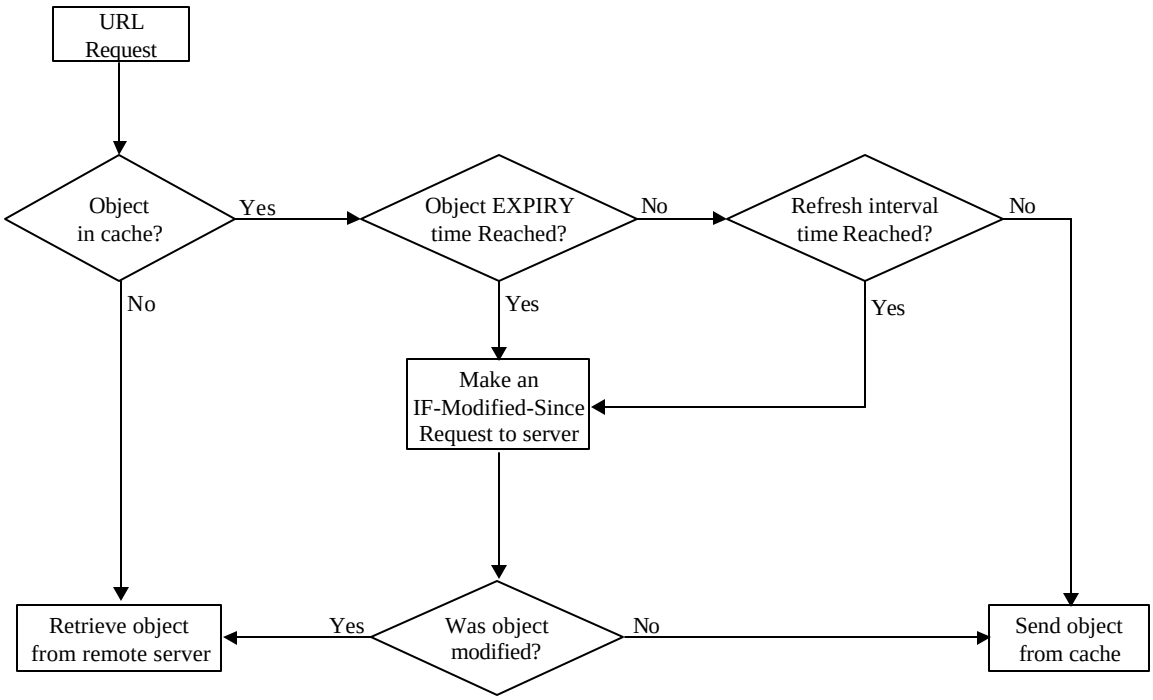

Figure 3: CERN Proxy cache logic [4]

Different from the approaches mentioned above, Wessels [4] used the HTTP server developed by CERN [9] as a proxy cache for the experiments. Figure 3 illustrates the combined usage of client polling and TTL approaches in the proxy cache logic. Beside the proxy cache, the author also developed a cache management program that runs in conjunction with the proxy cache to add, update and expire cache objects at regular intervals. The interesting feature of Wessels' experiments is a two-level cache, one called short-term, the other long-term. Which cache an object stays in is all controlled by the cache manager. Figure 4 illustrates the interaction of proxy/caching components of the experiment model.

[4] didn't compare different cache consistency approaches. Instead, the author developed the software, which relies heavily on the proxy for cache consistency, but takes advantage of invalidation when available to provide fewer stale objects. The experiment results show the dramatic improvement of user response time by using the proxy cache.

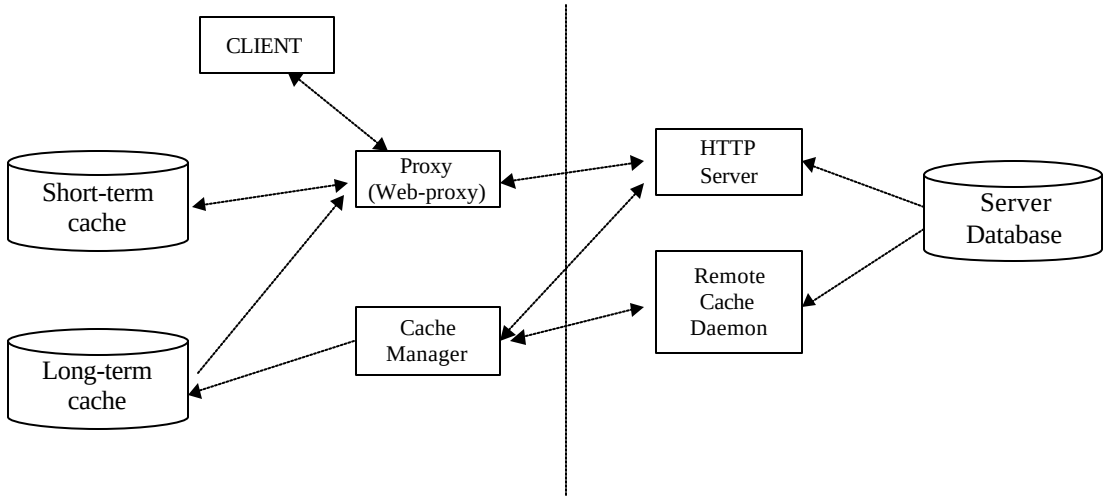

Figure 4: Interaction of Proxy/Caching components [4]

Wessles' experiment indicates that using a proxy cache improves user response time. The cache manager plays a crucial role in this model because only *it* is responsible for the cache consistency.

3.3 Limitations and Improvements

Both [2] and [3] tried to find the best cache consistency method by conducting experiments in the environment they set up. However, besides the limitations of the different methods, their experiments have certain degrees of limitations as well.

Weak cache consistency could be implemented where the object freshness demand from the end-user is not so strict. For example, it could be used for online newspapers that change daily. However, at the situation such as online weather forecast, or stock quote, where the requested document must always be up-to-date, a strong cache consistency mechanism has to be implemented.

The trace replay in [3] is performed in a local area network instead of the Internet. Under this environment, client response time is certainly much better than a real Web environment. The simulated environment is comparable to a high-speed, high-performance wide area network, but it cannot reflect the actual bandwidth of the Internet. The other point is that the experiments use server traces instead of client or proxy caches. Because the user requests seen by the server are partially filtered by client caches already, server traces show a lower hit ratio at the client sites. This means that in reality, polling-

every-time performs even worse than the results shown in [3]. In my opinion, I would not recommend polling-every-time because it sends validation message to web server whenever there is a user request, even though the requested object is in the cache. This is not an efficient way to reduce client latency, although it can reduce unnecessary file transfer across the network.

Update threshold is [2]'s favorite based on the experiment results. However, it is not easy to decide the individual update threshold value for each document. Currently the value is configured manually by the cache administrator, which creates a long distance from perfection. The same problem applies to TTL.

One major problem with invalidation approaches is that they are often expensive. This is because the server has to maintain a list of client caches that contain copies of certain documents. Once the document is changed, the server needs to send invalidation messages to all the client caches on the list. Not to mention the overhead for the server to send the control messages, the server might need fairly large storage space for such a list.

Another problem with invalidation is how to deal with failures. If the server crashes, the user request could still be responded with the cached document even though it might be stale. But if a client cache crashes, the server will not receive acknowledgement from the cache after it sends out invalidation messages. The server has to keep on trying, because otherwise the cache will never know that the document it caches is stale. If we take network partition into consideration, The time taken for the server invalidation message to reach the client might be rather long, even an extended period of time, during which the user will view stale documents without knowing it. [3] suggests that the only way to deal with such kind of problem is to have the client cache to contact the server from time to time to make sure that the network connection is OK and the server is up.

There are a number of ways to improve the current cache consistency approaches. In my opinion, adaptive TTL is pretty attractive, if the rate of stale hits could be kept under 5% or even lower. Therefore we could add some invalidation function to the server. In order to minimize the server storage overhead, we assume clients that request a certain document within 3 days are interested in being notified for invalidation. If the document is changed and right at the moment the server load is not high, it will send invalidation message out to those client caches that requested the document in the past 3 days. In this

way, there is not much overhead generated at server side, while the cache stale hit rate could be effectively reduced.

As strong advocates of invalidation approach, Cao and Liu [3] presented their improvement solution, which is called "two-tier-lease-augmented invalidation". First they add a "lease" field to all the documents sent from the server to a client cache. In this way the server promises to notify the client by sending invalidation message if the document changes before the lease expires. Meanwhile, the client promises to send an "if-modified-since" message to the server once the lease expires to validate the freshness of the document. In this way, the server doesn't have to remember all the client caches that keep a copy of the document, instead it only needs to remember clients whose leases have not expired yet, thus saves storage space on the server.

Second, for regular "get-object" requests, the server assigns a very short lease value (could be zero), and a regular lease to "if-modified-since" requests. In this way the server could filter out the client list and keep only those caches requesting to view the document for the second time. In this way the server storage overhead can be further reduced.

Cao and Liu also suggested improvement to their implementation in order to avoid the worst-case latency problem, by creating separate process to deal with sending out invalidation messages.

As an enhancement to weak cache consistency, A mechanism called pre-fetching could be used to reduce the number of stale documents forwarded to the end-user. It also reduces the delay resulting from reloading the new version of the document when it is requested. Pre-fetching is initiated by the cache server (probably proxy server). The documents are pre-fetched or more precisely "re-fetched" before they are requested. Usually, documents are pre-fetched because they are out-of-date or they will become stale in the near future.

Pre-fetching introduces both new traffic and additional processing to the cache server. A trade-off has to be made between the gain resulting from pre-fetching and its side effects on the network traffic and the cache server overload. The ideal implementation of pre-fetching should optimize two factors: the frequency at which the documents are pre-fetched and which documents are pre-fetched. In order not to overload the cache server, pre-fetching could be performed when the cache server is not busy with the users'

requests, for instance, during night hours where both the network and the cache are not submitted to heavy loads.

There are still other possible ways to refine the current cache consistency approaches for Web. Next section we will look into the cache consistency algorithms for client/server environment and try to come up with a feasible mechanism that could be used for the WWW.

4. Cache Consistency in Transactional Client/Server Environment

The web is fundamentally different from a distributed file system in its access patterns. Comparing to cache consistency issues in transactional Client/Server environment, those for the Internet might be much simpler because conflicting updates resulted from concurrently running transactions will never occur in a web environment. However, it is beneficial to study the consistency algorithms in client/server architecture in order to find some good approaches suitable for the Internet.

Figure 4 shows a reference architecture for a data-shipping client/server DBMS. The DBMS consists of two types of processes that are distributed throughout the network. First, each client workstation runs a Client DBMS process, which is responsible for providing access to the database for the applications running at the local workstation. Applications send database access requests to their local client DBMS process, which executes the request, in turn sending requests for transaction support and for specific data items to the Server DBMS processes. Server DBMS processes are the actual owner of data, and are ultimately responsible for preserving the integrity of the data and enforcing transaction semantics. The Server DBMS processes manage the stable storage on which the permanent version of the database and the log reside.

In a client/server environment, in order to reduce network traffic, data is cached at client site. But whenever there is cache technology, there comes the issue for consistency as well. Cache consistency protocols for client/server database systems have been the subject of much study in recent years and at least a dozen different algorithms have been

proposed and studied in the literature ([1], [5], [6]). [1] provides a taxonomy that categorizes most of the proposed transactional cache consistency algorithms based on whether they detect or avoid access to stale data. This is somewhere similar to the concept of weak and strong consistency in Web context. In fact, the similarity is not limited to category itself, but the way consistency is enforced and the time it is enforced. This survey will not go too much deep into each individual algorithm. Instead the focus of this section will be on how the algorithms are categorized, their similarity and differences from a conceptual point of view.

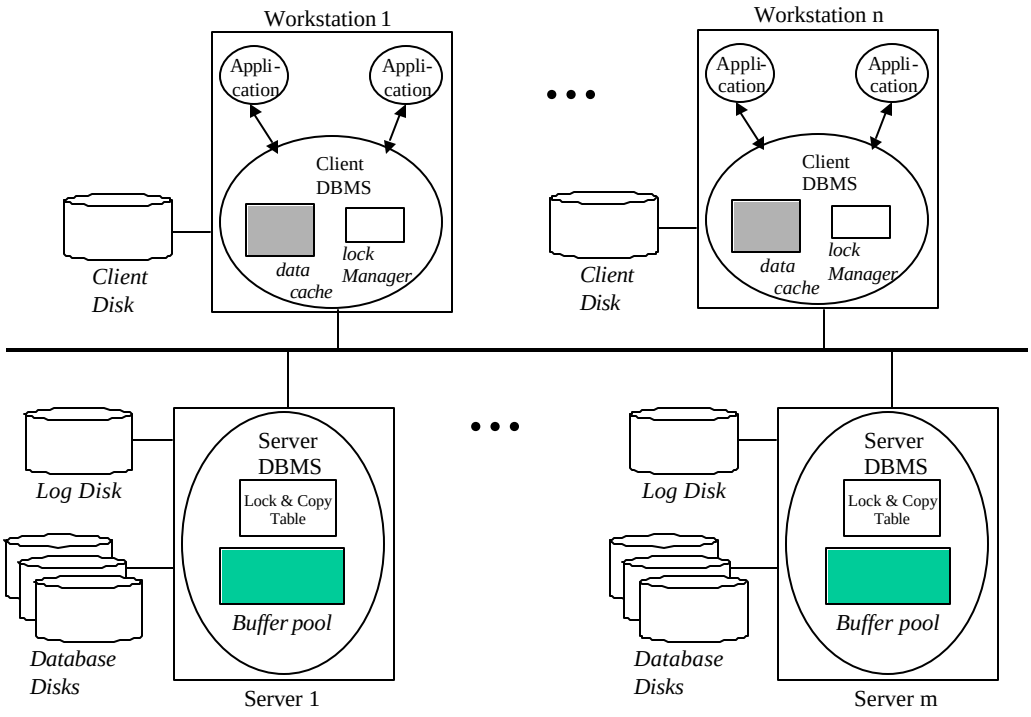

Figure 4: Reference architecture for a data-shipping DBMS [1]

Most cache consistency algorithms in client/server architecture could be categorized into *detection-based* or *avoidance-based*, depending on the choice of *Invalid Access Prevention* [1]. Algorithms that use *avoidance* for invalid access prevention ensure that at any time, all cached data is up-to-date. Those that use *detection* allow stale data to remain in client caches and ensure that transactions are allowed to commit only if it can be verified that they have not accessed such stale data.

Transactional cache consistency maintenance algorithms must ensure that no transactions that access stale data are allowed to commit. A little bit different from the Web context, in a transactional client/server environment, a data item is considered to be stale if its value is older than the item's latest committed value. The taxonomy in [1] partitions consistency maintenance algorithms into two classes according to whether their approach to preventing stale data access is detection-based or avoidance-based. Detection-based algorithms require a transaction to check the validity of accessed data before or at the point when the transaction commits, while avoidance-based algorithms ensure that stale data is removed from client caches as early as possible.

4.1 Detection-based Algorithms

Detection-based algorithms allow stale data copies to reside in a client's cache for some period of time. Transactions must therefore check the validity of any cached page that they access before they can be allowed to commit. The server is responsible for maintaining information that will enable clients to perform this validity checking. There are three levels of differentiation in the detection-based side of the taxonomy: *validity check initiation*, *change notification hints*, and *remote update action*.

- **Validity Check Initiation**. This level of differentiation is the coarsest, based on when the validity of accessed data is checked by the transaction. In order for a transaction to successfully commit, the validity of any accessed data must be checked before the commit actually happens. There can be three possible classes of validity checking strategies:
  - Synchronous, on each initial access to a page (cached or otherwise) by a transaction.
  - Asynchronous, with checking initiated on the initial access, but the transaction does not wait for the result of the check.
  - Deferred, until a transaction enters its commit processing phase.

  All these three classes ensure that once the validity of data is confirmed, it will remain valid till the end of the transaction. To guarantee this rule, the server must

not allow other transactions to commit updates to such items until a transaction that has received a validity guarantee finishes. This is obviously more complicated that Web environment, where under most cases only the Web server has the authorization to modify its objects.

These three classes provide a range from pessimistic (synchronous) to optimistic (deferred) techniques. Therefore they represent different tradeoffs between checking overhead and possible transaction aborts. The asynchronous approach is a compromise. It aims to mitigate the cost of interaction with the server by removing it from the critical path of transaction execution, while at the same time lowering the abort rate and/or cost through the earlier discovery of conflicts.

- **Change Notification Hints.** Since the communication with the server is always an expensive operation, designers of detection-based algorithms often use optimism to reduce this cost. Optimistic techniques are oriented towards environments in which conflicts are rare and the cost of detecting conflicts is high. However, for frequently accessed data, a more pessimistic approached is used to ensure the validity of the data. Such technique is called *change notification hints*. A notification is an action that is sent to a remote client as the result of an update that may impact the validity of an data item cached at that client. Purging or updating a stale copy removes the risk that a subsequent transaction will be forced to abort as a result of accessing it.

- **Remote Update Action.** This is the final level of differentiation in the detection-based half of the taxonomy, which is concerned with the action taken when a notification arrives at a remote site. There are three options here: propagation, invalidation, and choosing dynamically between the two. Propagation results in the newly updated value being installed at the remote site in place of the stale copy. Invalidation, on the other hand, simply removes the stale copy from the remote cache so that it will not be accessed by any subsequent transactions. After a page copy is invalidated at a site, any subsequent transaction that wishes to access the page at that site must obtain a new copy from the server. A dynamic

algorithm can choose between invalidation and propagation heuristically in order to optimize performance for varying workloads.

4.2 Avoidance-based Algorithms

Avoidance-based algorithms enforce cache consistency by making it impossible for transactions to ever access stale data in their local cache. These algorithms use a read-one/write-all (ROWA) approach to replica management, which ensures that all existing copies of an updated item have the same value when an updating transaction commits. All of the avoidance-based algorithms mentioned in [1] require that the server keep track of the location of all page copies. There are four levels in the avoidance-based half of the taxonomy: write intention declaration, write permission duration, remote conflict priority, and remote update action.

- **Write Intention Declaration.** While all of the avoidance-based algorithms use the same policy for handling page reads, they differ in the manner in which consistency actions for updates are initiated. When a transaction wishes to update a cached page copy, the server must be informed of this write intention sometime prior to transaction commit so that it can implement the ROWA protocol. A write permission fault is said to occur when a transaction attempts to update a page copy for which it does not possess write permission. The taxonomy contains three options for when clients must declare their intention to write a page to the server:
  - Synchronous, on a write permission fault.
  - Asynchronous, initiated on a write permission fault.
  - Deferred, until the updating transaction enters its commit processing phase.

  The tradeoffs among synchrony, asynchrony and deferral for write intentions are similar to those previously discussed for the detection-based algorithms: synchronous algorithms are pessimistic, deferred ones are optimistic, and asynchronous ones are a compromise between the two.

- **Write Permission Duration.** In addition to when write intentions are declared, avoidance-based algorithms can also be differentiated according to how long

write permission is retained for. There are two choices at this level of the taxonomy: write permissions can be retained only for the duration of a particular transaction, or they can span multiple transactions at a given client. In the first case, transactions start with no write permissions, so they must eventually declare write intentions for all pages that they wish to update; at the end of the transaction, all write permission are automatically revoked by the server. In the second case, a write permission can be retained at a client site until the client chooses to drop the permission or until the server asks a client to drop its write permission.

- **Remote Conflict Priority.** The third level of differentiation for avoidance-based algorithms is the priority given to consistency actions when they are released at remote clients. There are two options here: wait and preempt. A wait policy states that consistency actions that conflict with the operation of an ongoing transaction at a client must wait for that transaction to complete. In contrast, under a preempt policy, ongoing transactions can be aborted as the result of an incoming consistency action.

- **Remote Update Action.** The final level on the avoidance-based side of the taxonomy is based on how remote updates are implemented. The options here are the same as in the detection-based case, namely: invalidation, propagation, and choosing dynamically between the two. An important difference between remote update actions under the avoidance-based algorithms and under the detection-based ones is that in the avoidance-based case, the remote operations are initiated and must be completed on behalf of a transaction before the transaction is allowed to commit. This is necessary to maintain the ROWA semantic guarantees that provide the basis for the correctness of avoidance-based algorithms. Therefore, if update propagation is used, all remote sites that receive the propagated update must participate in a two-phase commit with the server and the client at which the transaction is executing. In contrast, invalidation does not require two-phase

commit, because in this case, data is simply removed from the remote client caches.

There are many different ways to categorize various cache consistency algorithms for transactional client/server environment. The taxonomy presented in [1] is just one of them.

5. Conclusion and future work

Caching is currently the primary mechanism for reducing the latency as well as bandwidth requirements for delivering Web contents. This survey has compared weak cache consistency and strong cache consistency algorithms that are used in the Web environment, from conceptual point of view as well as the experiment results from various research papers ([2], [3], [4]). It also discussed a taxonomy that categorizes different cache consistency algorithms that are applied under transactional client/server environments. Although in the Web environment, there is no need to consider transactional commitment, we could still perform the consistency check asynchronously in order to leverage the server load, which is one of the possible ways to apply transactional cache consistency algorithms to Web environment. We could also apply "change notification hints" method to cached documents based on their popularity. I guess this is how the idea of adaptive TTL came from.

WWW is simply another huge client/server environment. Each computer an end user uses is just a small node within the Web. In order to access contents on a certain Web server, the user request might first go through a proxy server, or even a wide area network that connects this small node to the Web. Although the situation of WWW might be more complicated, one good news is that it is not transactional, i.e., the Web server doesn't need to worry about whether a client transaction is about to commit, or grant write permission to the client.

A choice has to be made whether to use weak or strong cache consistency, i.e., for the cache, whether to ensure the document freshness all the time, or just validate the objects after certain time interval. In my opinion, we could combine these approaches to maximize their advantages. I plan to study further into web caching algorithms, possibly

from those caching algorithms for a distributed client/server environment, and apply them to fit in the Web environment. If an algorithm could perform well in a general distributed environment, it could probably be a good candidate for the Web as well. There are also possible directions such as the two-level-lease cache consistency algorithm introduced in [3], and pre-fetching techniques that could possibly be implemented on the cache server side. This should be an interesting subject for research.

Reference

[1] Michael J. Franklin, Michael J. Carey and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Transaction on Database Systems,* 1997.

[2] James Gwertzman and Margo Seltzer. World-Wide Web Cache Consistency. *International Conference USENIX*, San Diego, CA, 1996

[3] Pei Cao and Chengjie Liu. Maintaining Strong Cache Consistency in the World-Wide Web. *Proceedings of the 17th International Conference on Distributed Computing Systems (ISDCS '97)*, 1997

[4] Duane Wessels. Intelligent Caching for World-Wide Web Objects. *International Conference of the Internet Society (INET),* Honolulu, HI, 1995

[5] Yongdong Wang and Lawrence A. Rowe. Cache consistency and concurrency control in a client/server DBMS architecture. *Proceedings of the ACM SIGMOD Conference on Management of Data* (Denver, CO), 367-377, May 1991.

[6] Kevin Wilkinson and Marie-Anne Neimat. Maintaining Consistency of Client-Cached Data. *Proceedings of the Conference on VLDB* (Brisbane, Australia), 122-134, 1990.

[7] A. Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems. *Proceedings of International Conference on Data Engineering*, New Orleans, Louisiana, March 1996.

[8] V. Cate. Alex – A Global File system. *Proceedings of the 1992 USENIX File System Workshop*, Ann Arbor, MI, 1-12, May 1992

[9] A. Luotonen, H. Frystyk and T. Berners-Lee. W3C httpd. *http://www.w3.org/hypertext/WWW/Daemon/Status.html.*