



Evaluation of Strong Consistency Web Caching Techniques

L. Y. CAO and M. T. ÖZSU

{y2cao,tozsu}@uwaterloo.ca

University of Waterloo, School of Computer Science, Waterloo, ON, Canada N2L 3G1

Abstract

The growth of the World Wide Web (WWW or Web) and its increasing use in all types of business have created bottlenecks that lead to high network and server overload and, eventually, high client latency. Web Caching has become an important topic of research, in the hope that these problems can be addressed by appropriate caching techniques. Conventional wisdom holds that strong cache consistency, with (almost) transactional consistency guarantees, may neither be necessary for Web applications, nor suitable due to its high overhead. However, as business transactions on the Web become more popular, strong consistency will be increasingly necessary. Consequently, it is important to have a comprehensive understanding of the performance behavior of these protocols. The existing studies, unfortunately, are ad hoc and the results cannot be compared across different studies. In this paper we evaluate the performance of different categories of cache consistency algorithms using a standard benchmark: TPC-W, which is the Web commerce benchmark. Our experiments show that we could still enforce strong cache consistency without much overhead, and Invalidation, as an event-driven strong cache consistency algorithm, is most suitable for online e-business. We also evaluate the optimum deployment of caches and find that proxy-side cache has a 30–35% performance advantage over client-side cache with regard to system throughput.

Keywords: WWW, Web caching, TCP-W

1. Introduction

The growth of the World Wide Web (WWW or Web) and the parallel increase in business activity over the Web have increased the load on resources. Most of the Web applications rely on Web caching to reduce network and server loads and client latency. Generally speaking, Web caching is a mechanism to store Web objects (such as HTML pages and image files) at certain locations for convenient future access. However, caching of objects at various points on the Web raises *cache consistency* as a prominent issue. If the original object on server is changed while the end user keeps accessing an out-of-date copy from its local cache, stale data would be accessed. On the other hand, the straightforward solution of directing every request to origin server totally discards Web caching and has adverse performance effects. In the past decade, significant effort has been directed at this problem and various caching architectures and cache consistency algorithms have been developed. In this paper we compare several representative consistency algorithms under the Web commerce environment and evaluate their performance using TPC-W [11], the Web commerce benchmark. Based on our experimental results, we draw conclusions

on the most suitable cache consistency mechanism for electronic commerce and the most efficient place to deploy a cache system. We focus on one class of algorithms known as strong Web caching consistency algorithms.

1.1. Problem specifics

Web caching addresses the issue of designing high performance Web sites for business activities over the Internet. In the context of WWW, caches act as intermediate service systems that intercept the end-users' requests before they arrive at the remote server. If the requested object is available in the cache, then it is returned to the user; otherwise the cache manager forwards the request on behalf of the user to either another cache or to the origin server. When the cache manager receives the object, it keeps a copy in its local storage (i.e., cache) and forwards the object to the user. The copies kept in the cache are used for subsequent user requests. Finding (not finding) copy in the local cache is referred to as a *cache hit (cache miss)*.

A Web cache may be used within a Web browser (client), within the network itself (typically on *proxy servers* that might be located between a department and an enterprise network, between an enterprise network and the Internet, or on a link out of a country), and at servers. However, some classes of documents usually cannot be cached, such as scripts and pay-per-view documents. Our research focuses on client caching (including proxies), and, unless otherwise noted, references to cache and cache manager should be interpreted as client cache and client cache manager.

Web caching has three main advantages [12]: reduced bandwidth consumption (fewer user requests and server responses that need to go through the network), reduced server load (some of user requests could be served locally), and reduced latency (because a requested Web page is cached, it is available immediately, and it locates closer to the client being served). Sometimes a fourth advantage is added: higher reliability, because some objects may be retrievable from cache even when the original servers are not reachable. Another positive side-effect of Web caching is the opportunity to analyze the usage patterns of organizations [12].

A major issue with caching is stale data access, i.e., the potential of using an out-of-date object stored in the cache instead of fetching the current object from the origin server. If the cache manager serves a client with its cached object whose original copy has changed, the client gets a stale copy. Always keeping the cached copies updated is possible if we keep contacting the server to validate their freshness. However, this means more control messages sent over the network, which consumes bandwidth and adds to server load, not to mention that the client might experience longer response time (usually referred to as *client latency*).

Therefore, the tradeoff is, either sacrifice document freshness for faster response time and fewer control messages, or enforce the consistency of cached objects with their original copies on the server by sending control messages, using other time-based mechanisms, or making the origin server to take full responsibility. The first is known as *weak cache consistency*, while the second is referred to as *strong cache consistency*. For every online

application system, a decision has to be made whether to maintain weak or strong cache consistency. In this paper, our focus is on strong cache consistency algorithms.

1.2. Case for strong consistency

To some extent, the literature agrees that always keeping the cached copy consistent with the original object means more latency observed by the client, and possibly more network traffic. For this reason, many caching systems apply weak cache consistency, believing that methods such as Time-To-Live (TTL) are sufficient and most appropriate for Web caching [6].

For many Internet services, strong consistency is not required, although it may be desired. One good example is online newspaper and journals, where servicing somewhat outdated information may be tolerable. However, for many e-business applications, weak cache consistency might not only be unsatisfactory and even annoying, sometimes it will be completely unacceptable. In the following, we highlight some applications which require strong cache consistency.

1.2.1. Online bookstore. Today, like many other businesses, bookstores are extended to the Internet. Good examples of online bookstores include Amazon, Chapters, etc. Consider a customer of one of these bookstores (could be an individual or business organization) who wants to buy 3 copies of a book, and checks the available amount from Web page which shows that there are 5 available copies. The customer might happily place the order and go ahead with the purchase. If the available copy number is fetched from the local cache instead of Web server, this number could be the available amount of several minutes or even hours ago. In that case, if the book is popular, it is quite possible that at the time the user checks its availability, the book has only 1 copy left or it is already out of stock. Therefore the customer is prevented from completing the transaction, causing dissatisfaction and lost business.

1.2.2. Stock quote. It is now possible to get stock quotes online via the services provided by online brokerage companies. Many people make buy/sell decisions based on the information they receive from the Internet. If stock prices are not pushed, but pulled by a user, and if there is a cache that services user requests from its cached copies, it is possible that such information is out-of-date. Here the strict demand for such information is that the information must always be updated. The result of such stale information because of weak cache consistency could be serious.

1.2.3. Online auction. This is another good example where strong cache consistency must be applied if Web caching mechanism is used. Each auctioned item comes with a current bid, a closing time, and bid increments. Any registered customer can place a bid before the closing time, but only those bids that are higher than the price at the time the bids are places will be valid, otherwise they will be rejected. In this case, accessing stale data that is provided by cache will result in failure of bidding on the item. Weak cache consistency is completely intolerable in this case.

1.3. Paper scope, contributions, and organization

As discussed in previous sections, there are both desirable features and undesirable drawbacks of applying either weak or strong cache consistency. Even though there are reported performance studies on the various techniques, these are ad hoc and do not follow standard evaluation methodologies. In this paper we evaluate various cache consistency algorithms under electronic commerce environment because we believe that strong cache consistency is a critical prerequisite for any successful online business model. TPC-W Benchmark is used to generate workloads to mimic an online bookstore. In this paper, we first come up with a two-dimensional classification of cache consistency algorithms, then focus on more representative ones. These algorithms are: Time-To-Live, Invalidation, Polling-Every-Time and Lease. For each algorithm, we deploy it both at client-side cache and proxy-side cache and compare the performance results. We also implemented infinite caching mechanism, because we want to use it as our upper-bound performance reference of system throughput and response time.

The paper is organized as follows. In Section 2, we review the fundamental cache consistency techniques that are considered in this paper. The experimental setup, including a description of TPC-W and the specifics of the setup used in this paper, are discussed in Section 3. The experimental results are presented and discussed in Section 4. Our conclusions and plans for future work are included in Section 5.

2. Cache consistency techniques

Any caching system, regardless of the caching architecture it deploys or how the objects are cached, has to deal with a fundamental issue: the consistency between objects in cache and their original copies. In order for caches to be useful, consistency must be enforced, i.e., cached copies should be updated when the objects get changed on origin server. There are many ways to maintain consistency, with different loss and gains, and many cache consistency algorithms have been designed and implemented. Based on the role of the client and the server in the cache consistency control processing, we could have three categories of consistency algorithms: *Client Validation*, *Server Invalidation*, and *Client/Server (C/S) Interaction*. In client validation approach, it is the client (or proxy) cache manager that is responsible for verifying the validity/freshness of its cached objects. With server invalidation, caches always assume the freshness of the objects they cache, and whenever an object is changed on a Web server, it is the server's responsibility to notify all the caches who cache that object to delete their stale copies. However, the way that the server invalidates stale copy could vary. In the C/S interaction category, the client and the server work interactively to enforce consistency to certain level as required by application.

From another point of view, based on how strict the caches are kept consistent, algorithms could be classified into two categories: *strong cache consistency* and *weak cache consistency*. We have briefly discussed these two terms in Section 1. As defined in [2], weak consistency is the model in which a stale copy of document might be returned to the user, while in strong consistency model, the consistency between cached copies and

Table 1. A classification of cache consistency algorithms

	Client Validation	Server Invalidation	C/S Interaction
Strong	Polling-every-time	Invalidation	Lease
Weak	TTL, PCV	PSI	N/A

original ones is always enforced, and no stale copy of the modified document will ever be returned to the user. We can see that validation/invalidation classification and weak/strong taxonomy are orthogonal although overlapping.

Table 1 gives a two-dimensional classification of most publicly known cache consistency algorithms. In the category of “C/S interaction-weak,” there are no current algorithms. We discuss the strong consistency algorithms in more detail in this section, since these are the foci of our work. We also describe a weak consistency algorithm (Time-To-Live), since it is very commonly deployed and we use it for baseline comparison.

2.1. Time-To-Live

Under Time-To-Live (TTL) approach, each object (document, image file, etc.) is assigned by origin server a time-to-live (TTL) value, which could be any value that is reasonable to the object itself, or to the content provider. This value is an estimate of the cached object’s lifetime, after which it is regarded as invalid. When the TTL expires, the next request for the object will cause it to be requested from the origin 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 (IMS) 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 response or explicitly specifying it at object creation time. HTTP protocol version 1.1 contains header keywords such as “expires” and “max-age” to notify the cache manager how long the object could be deemed valid. However, a large number of HTTP responses do not have any expiry information [10], which forces the cache manager to use heuristics to determine the object lifetime.

The challenge in this approach lies in selecting an appropriate TTL value, which reflects a tradeoff between object consistency on the one hand, and network bandwidth consumption and client latency on the other. If the value is too small, after every short period of time the cached copy will be considered stale. Therefore, many IMS messages will be sent to origin server frequently for validity check, which results in extra network traffic (although it might be trivial compared to the actual file transfer, if the file size is big) and server overhead. By choosing a large TTL value, a cache will make fewer external requests to origin server and can respond to user requests more quickly. Cached objects are retrieved as updated versions although their original copies are already changed on the server, thus, out-of-date document may be returned to the end-user. In [6], the authors initially use a flat lifetime assumption for their simulation, which means that they assign all objects equal TTL values. This is also called *explicit TTL*, which results in poor performance. This has later been modified with the assignment of the TTL value based on the popularity of

the file. This method is also mentioned in [2], where this improved approach is termed *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. This heuristic estimation traces back to the Alex file system [3]. Gwertzman and Seltzer also mention that globally popular files are the least likely to change. By using adaptive TTL, the probability of stale documents is kept within reasonable bounds ($<5\%$) [6].

TTL algorithm and its variations all enforce weak cache consistency, since it is possible that the cache manager satisfies client requests with stale objects. Once an object is cached, it can be used as the fresh version until its time-to-live value expires. The cache manager assumes its validity during the TTL period. The origin server does not guarantee that the server copy of an object will remain unchanged before the object's TTL expires. In other words, server is free to update the object even though its time-to-live value has not expired yet, thus making it possible for user to get out-of-date objects. On the other hand, applying TTL algorithm has the advantage that the origin server does not need to notify caches when the objects are changed on server.

One important improvement of TTL is to modify it so that each communication with the server updates the validity of the cached objects. Piggyback Client Validation (PCV) [7] achieves this by attaching to each message to the server, a list of cached objects whose original copies are on that server. These objects are not necessarily requested by end user at the time, instead they are in the piggyback list because their TTL values have expired. Therefore, the validity of these objects are checked in advance, which reduces the possible stale hit ratio. Meanwhile, using a batch of validation messages could reduce the control message overhead.

2.2. Client Polling and Polling-every-time

Both *Client Polling* and *Polling-every-time* belong to the category of Client Validation approach, i.e., client (either browser or proxy cache) is responsible for checking the validity of cached objects. With client polling approach, the client (cache) periodically checks back with the server to determine if cached objects are still valid. It is somewhat like adaptive TTL, because under both cases the client sends out validation message from time to time to check if the object is still valid. Alex FTP cache [3] uses an update threshold to determine how frequently to poll the server. The update threshold is expressed as a percentage of the object's age. An object becomes invalid when the time since last validation exceeds the product of update threshold and the object's age [6]. 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 ($10\% \cdot 30$ days). Since the object was checked the day before, requests that occur during the next two days will be satisfied locally, and there will be no communication with the server including control messages. 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.

Obviously, client polling cannot guarantee that the object stored in cache is fresh, because cache manager determines object validity by time-to-live value or certain heuristic estimate, which might not be accurate at all. Therefore, client polling (some researchers also call it periodic polling) is only a weak form of consistency control. By contrast, polling-every-time enforces strong cache consistency by sending If-Modified-Since (IMS) messages to origin server to confirm object validity whenever a user request arrives and an object is present in cache. However, sending polling message upon each cache hit adds too much traffic to the network which is not favorable at all. Meanwhile, it might incur unnecessary delay in response, because when a cache hit occurs, even though the cached object is still valid, the consistency algorithm still requires the cache manager to confirm the object validity with origin server. Polling-every-time is generally ranked as an undesirable consistency mechanism, mainly because it does not take full advantage of Web cache. Even though all the cached objects are fresh copies of original, the end user still has to experience the validation delay.

2.3. Invalidation

The Invalidation algorithm frees client cache manager from the burden of sending IMS messages. To ensure strong consistency, if an object *A* gets changed on the server, the server must send out an invalidation message right away to all the caches that store a copy of *A*. In this way, cache managers do not need to worry about object validity. As long as an invalidation message is not received, the object is valid. Invalidation approach requires the server to play a major role in the consistency control process over cached objects. This might be a significant burden for the Web server, because it has to maintain state for each object that has ever been accessed, as well as a list of the addresses of all the requestors. Such a list is likely to grow very fast, especially for popular objects. The server has to maintain at least a big storage space for keeping the lists. On the other hand, although an object is stored in a cache whose address is kept on the server list, this object might be evicted from the cache later on, because it is rarely if ever requested again, or the cache manager needs free space for newly-arrived objects. Therefore, it does not make sense for the server to keep the address of that cache on the list. Even worse, if the object is about to be changed, the server has to send invalidation message to the caches whose addresses are on the list, but they no longer keep the object, which adds unnecessary traffic to the network.

Since invalidation approach tends to make sure that strong cache consistency is always enforced, every invalidation message must be acknowledged. A server will not update an object until it receives all the acknowledgement messages from the caches to whom it sent invalidation notices. This is sometimes called *delayed update* [10]. However, this approach faces availability problems, since the unreachability of one cache causes the server to keep waiting. None of the caches will get updated version of the object, which results in stale access by end user. Yu et al. argue that it is absolutely unnecessary to delay updates on Web pages because if the need to update a page occurs, the page itself has changed semantically

anyway, no matter whether or not the server updates it [15]. So the cached copy of the page is out-of-date regardless of any update delay.

There are possible modifications that alleviate the problems of invalidation. For example, invalidation messages can be multicast to reduce server burden. The basic idea, as discussed in [10], is that a multicast group is assigned to each object. When a cache manager requests a certain object, that cache is added to that object's multicast group. Therefore, there will be a list of cache addresses for each multicast group. Such lists are stored in the multicast membership state maintained by routers. When an object update takes place, the server just sends one invalidation message to the corresponding multicast group, and then it is the router's job to multicast the invalidation message to all the caches that belong to the multicast group.

Providing strong consistency using invalidation is indeed very difficult given a large and widely distributed system such as the Internet. The heterogeneous nature of the Internet makes the time delay from server to proxy and from proxy to end user unpredictable. It is also quite possible that a cache, once connected to the Internet and communicates smoothly with other resources, crashes or disappears all of a sudden. The server's operation is more complicated if it sends out an invalidation message to such an unavailable client and keeps waiting for acknowledgement from that client before updating Web pages.

A variation of sever invalidation is Piggyback Server Invalidation (PSI) [8], which sends invalidation messages in batch compared to individual invalidation notification. As long as strong consistency is not the major concern, this performs better than synchronous invalidation in a sense that it reduces the number of invalidation messages in the system.

2.4. *Lease-based techniques*

The concept of *lease* was introduced as early as 1989 by Gray and Cheriton, in their paper addressing the cache consistency issue in a distributed file system [5]. Leases could also be applied in Web caching to provide stronger cache consistency control while solving the issues that seem insurmountable for the algorithms discussed in previous sections. The basic idea is that when the origin server sends out an object in response to a client request, it assigns a "lease" value to the object. The server promises not to update the object before the lease expires. The cache then is ensured that as long as the lease has not expired, the object is valid and any request for that object will be a positive hit. When the lease expires, upon next request of the object, the cache manager will send IMS messages to origin server, and the server either responds with the new version of the object, or, if the object has not been changed yet, extends the lease and returns that to the client, and the same rule applies.

Lease algorithm maintains strong cache consistency while keeping servers from indefinitely waiting due to a client failure. If a server cannot contact a client, it delays updating the object until the unreachable client's lease expires, and from then on it becomes the client's responsibility to contact the server for validation. On the other hand, Lease algorithm needs to be implemented both at client side and on the server. That is why we put it into the "C/S interaction-Strong" category.

As with TTL algorithm, the lease duration affects the efficiency of the algorithm itself. If the lease value is shorter than the interval between two requests, every subsequent request comes when the current lease has already expired. In this case lease becomes polling-every-time, which is far from desirable. However, this does not mean that a longer lease is better. Having a very long lease out there forces server to delay object updates until that lease expires. To solve this problem, Yin et al. introduce *volume lease* in addition to *object lease* that goes with each individual object [13,14]. In this approach, each volume lease is assigned to a set of related objects on the same server. In order to use a cached object, a client must hold the leases on both the object and the volume it belongs to. The cache manager cannot respond to a user request with cached object unless both the object and volume lease on that object are valid. Server is free to update an object as soon as either the volume or object lease on the object has expired. By making object leases long and volume leases short, server can make object updates without long delays. Meanwhile, long object leases prevent the cache manager from having to validate individual objects frequently.

3. Experimental setup

In this section we describe the experimental setup that we use in our study. As indicated earlier, we use TPC-W benchmark as our main tool. This both constrains our workloads and the type of environments we consider. TPC-W is described in Section 3.1. We use a pure Java implementation of TPC-W done by a team at the University of Wisconsin-Madison. A second issue we discuss in this section is the particular system model that we adopt based on TPC-W (Section 3.2). Finally, the system parameter settings are presented in Section 3.3.

3.1. TPC-W benchmark

TPC-W is a transactional Web benchmark [11] that simulates the activities of a 24/7 online bookstore. The benchmark measures both business-to-consumer (B2C) and business-to-business (B2B) activities. It includes real world features such as security, shopping carts, credit card validation, load balancing and Web page information from the database.

In order to model varying bookstore sizes, the benchmark permits database size of 1,000 to 10,000,000 book items in tenfold increments. The number of books is defined as the *scale factor* of the benchmark, which is discussed in more detail in the following section.

The benchmark defines 14 Web interactions that normally occur in an online bookstore Web site. The emulated browser (EB), as we will elaborate in Section 3.3, will go through one or more of these interactions just like a normal customer would while visiting the online store. For example, a customer can browse pages containing a list of new arrival or best selling books, or search for a certain book item by title, subject or author name. A product page will give the customer detailed information for the book along with a picture of the book's front cover. The customer may then place an order for books through the order pages, including credit card verification. If the customer is new to the bookstore,

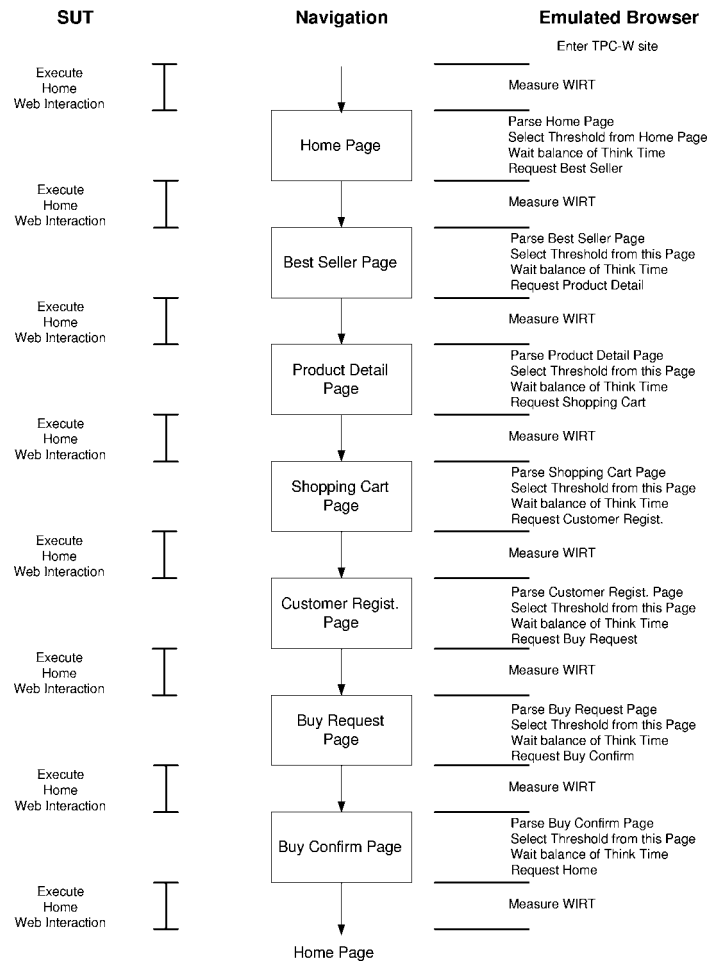


Figure 1. An example of EB activities at TPC-W Web site (adapted from [11]).

the Web site will ask him/her to fill out a customer registration form, and the information will be stored at database server. If the customer has visited before, the information will be retrieved from database and filled in the order form automatically. Shopping cart is an important component of online transaction processing, and the customer is free to add or delete items from the cart. The customer is also able to review the status of previous orders at any time. Each of the 14 Web interactions is represented as a set of Web pages on the bookstore Web site. Figure 1 gives an example of how an EB might move through the TPC-W Web site, and the interactions between the system under test (SUT), which is the collection of servers that support TPC-W e-commerce interaction, and EB.

Obviously, the fourteen Web interactions impose different workloads on the Web server, because each of them demonstrates a different kind of user operation, from read-only to

updating database information. The variety enables the benchmark to test the performance of different components in the SUT.

The users who come to visit the bookstore Web site might have different interests. Some might be just browsing, searching for books and comparing prices. Others would have already made up their minds and are ready to buy. The interactions they will go through would be different too. In order to model different types of users, the benchmark defines two categories of Web interactions: browsing and ordering. Browsing interactions include displaying the home page, searching for an item, viewing product details, etc., while ordering interactions include the more resource intensive ones such as buying an item, updating items in the shopping cart, displaying an order, etc. Based on the ratios of these two categories, the benchmark measures the Web Interactions Per Second (WIPS) for three different types of mixes, namely Shopping (WIPS), Browsing (WIPSB) and Ordering (WIPSO). The Shopping mix (WIPS) models an 80%–20% ratio between the browsing and ordering interactions and is intended to emulate a typical user's shopping activity. The Browsing mix (WIPSB) models a 95%–5% ratio and emulates “window shoppers” who spend most of their time browsing the online store and seldom purchase anything. Conversely, the Ordering mix (WIPSO) has a 50%–50% ratio for both categories and emulates “power buyers” shopping at the online store. This mix attempts to mimic a B2B type of workload.

More than 90% of all the Web interactions at the online store are dynamic in nature, which means that the Web pages generated by the interactions are put together on the fly, using dynamic data retrieved from multiple sources. Naturally, caching such data will be difficult, and more detailed discussion can be found in the next section.

3.1.1. System architecture. TPC-W environment consists of a number of components: the browser emulator, the system under test, and the payment gateway emulator. We describe these below.

3.1.1.1. Remote Browser Emulator. The Remote Browser Emulator (RBE) is the software component that drives the TPC-W workloads. The purpose of the RBE is to drive the SUT by simulating users using the site to examine and purchase books. Each simulated user is called an *emulated browser* (EB). The RBE manages a collection of EBs. We follow the notion in [11] that the term RBE includes the entire population of EBs that it manages. Essentially, each EB sends out HTTP requests, as a Web browser would, and receives the HTML response from the Web server. Based on the content received EBs randomly make the next request, emulating the behavior of typical users.

By managing EBs, the RBE needs to make sure that for each user session that an EB measures, a specific duration of user session is maintained. This is to ensure that EBs are really emulating the browsing and/or shopping activities of a real Web user. This is called User Session Minimum Duration (USMD). For each User Session, the EB generates a USMD randomly selected from a negative exponential distribution. Like a normal Web user, the EB can start a new user session right after it terminates one. This is controlled by RBE, too.

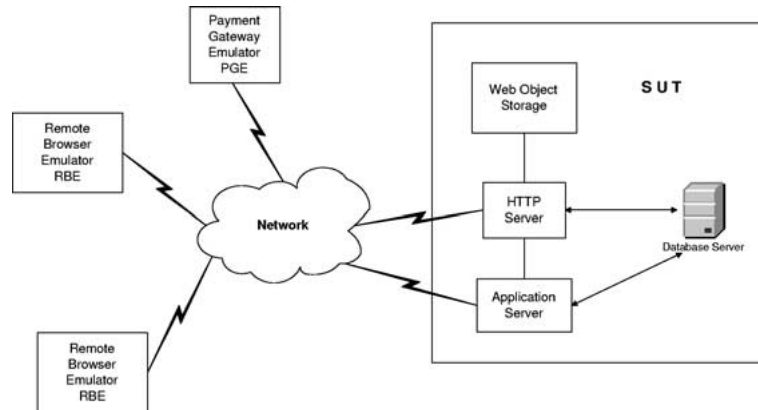


Figure 2. Model of the complete tested system (adapted from [11]).

3.1.1.2. System Under Test. The System Under Test (SUT) is the integrated part on which we conduct performance evaluation. As displayed in Figure 2, this part consists of Web server, application server, database server, as well as Web objects such as image files that are stored on server file system. The network interface card, which is required to form the physical TCP/IP connections, is also regarded as part of SUT.

3.1.1.3. Payment Gateway Emulator. The Payment Gateway Emulator (PGE) represents an external system that authorizes payment of funds as part of purchasing transactions. This part usually includes client message encryption, generating authorization code and establishing secure socket layer (SSL) session for authorization security check, as well as returned message decryption. Although this is a critical component of real e-commerce deployment, it is not critical to our study as long as the delay is properly modelled. That is what is done in this study, assuming a positive acknowledgement from PGE.

3.1.2. Scaling requirements. The intent of the scaling requirements is to maintain the ratio among the Web interaction load that is experienced by the SUT, the size of the tables accessed by the interactions, the requirement space for storing related information, and the number of EBs generating the transaction load. The throughput of the TPC-W benchmark is driven by the activity of the EBs, each of which emulates exactly one user session at a time. In order to increase throughput demand on the SUT, the number of EBs has to be increased. Obviously, the configured EBs must remain active and generate Web interactions throughout the entire measurement interval.

Besides the number of EBs, database size is another scaling factor that will affect throughput. According to the benchmark specification, valid database sizes are 1,000, 10,000, 100,000, 1,000,000 and 10,000,000 book items. TPC-W results can only be compared at the same scale factor level.

There are altogether 8 relations that are required by TPC-W: *Customer*, *Country*, *Address*, *Orders*, *Order_Line*, *Author*, *CC_Xacts* and *Item*. The number of records in *Item* table is explicitly specified as one scale factor of the database,

i.e., from 1,000 to 10,000,000, representing an online bookstore from very small size to a significantly large one. For each of the emulated browsers, the database must maintain 2,880 customer records and associated order information, and the sizes of most tables are determined by either the number of EBs or the number of customers.

The reported WIPS throughput is required to satisfy the following relationship:

$$\frac{\text{number of EBs}}{14} < \text{WIPS} < \frac{\text{number of EBs}}{7}.$$

According to the specification, the intent of this requirement is to prevent throughput that exceeds the maximum, where the maximum throughput is achieved with infinitely fast Web interactions resulting in a null response time and minimum required think times. In fact, from our experiments, we found that it is very difficult to get even close to the upper bound, given the fact that we used a positive acknowledgement generator to emulate the PGE. If a full-functioning PGE is in place, we believe the resulting WIPS will be closer to the lower bound. However, some WIPS results in our experiments are below the lower bound. The reason is that we added the delay at network transfer and application server side. The limited processing capability of our application server makes the server delay inevitable. The processing at cache side takes time, too, which affects WIPS as well. Setting up the lower bound helps the tester to verify whether the SUT has been over-scaled.

3.2. System model

We set up our model in accordance with TPC-W benchmark requirements. At the same time, we add application server queuing manager component to the model to reflect application server delay. Figure 3 is an overall picture of our system model. In this figure, server-side components correspond to SUT while client-side components correspond to RBE with the addition of a proxy server.

3.2.1. Client. According to the benchmark specification, the RBE acts as the driving program for the benchmark workload. It creates multiple threads during runtime, which simulate real Web users who browse the online bookstore Web site and place orders to buy books. The user thinktime is calculated by the RBE. Individual EBs are not allowed to communicate or share information directly with each other.

3.2.2. Proxy server. Proxy server is not mandatory for an e-commerce Web site, because Web interactions do not need to go through proxies. However, the proxy server component is included in this study for the following reasons. First, most individual customers access the Internet through ISPs, which act materially as proxies. Second, a RBE acts exactly like a proxy server to all the EBs it manages. The inclusion of a proxy server component gives RBE a physical location in the architecture. However, we did not implement a full-fledged proxy server, rather the implementation is restricted to a proxy-side cache together with RBE. The measurement interval of the benchmark is set and checked by RBE as well. Once the measurement period is up, the RBE forces all currently-running

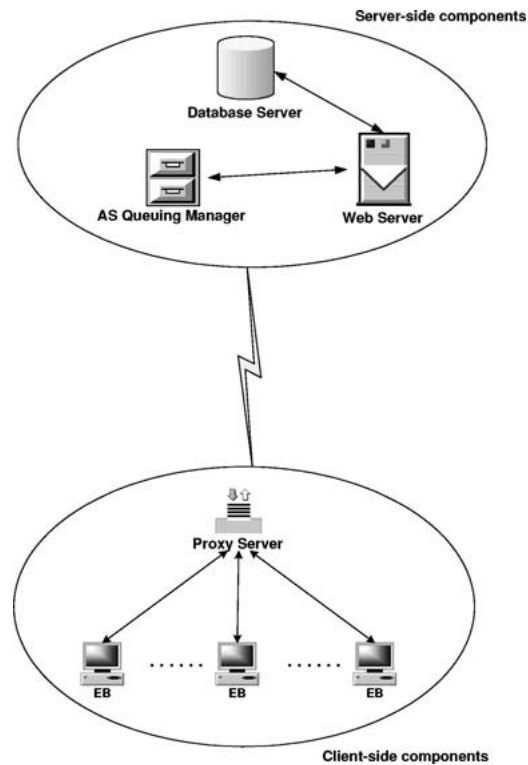


Figure 3. Simulation system model.

emulated browsers to stop and produce statistics reports based on information gathered during the measurement interval.

3.2.3. Database server. We use IBM DB2 Universal Database Version 7.2 (Enterprise Edition) as the database server. We used the default installation configuration, but changed one parameter, a DB2 variable called `maxappls`, which is the maximum number of active applications allowed by DB2. The default value is 40, which we changed to 1000. Since we are simulating the real world Web behavior, there can be a lot more than 40 concurrent users and each user might try to connect to DB2 independently and simultaneously. When we populated the database, we assumed that there are at most 100 concurrent users accessing the online bookstore, and the bookstore sells 100,000 book items.

3.2.4. Web server. In this study Jakarta Tomcat 3.2.1 is used as the Web server. Although it is not a fully functioning HTTP server, it is sufficiently powerful for this study, partly because it supports servlets.

According to the nature of the Remote Browser Emulator, the main program creates multiple threads, each of which simulates a real Web user who performs browsing or shopping activities in an online bookstore. Upon each URL request that is generated by an

emulated browser (EB), there is a servlet request to the Web server. A connection is set up between the client and the Web server.

3.2.5. Application server. Application server is a critical part of an e-commerce Web site. Generally speaking, it contains process flow and logic of the business. It also has security mechanisms that prevents theft of private customer information by hackers. The application server is also responsible for interacting with external system which authorizes payment of funds as part of the purchasing transactions.

The Java implementation of the benchmark that is used does not have the application server part. Instead, it contains a set of servlets that respond to Web user requests with corresponding information. To some extent, this could be regarded as application server, but it does not model the response time delay and concurrent service capability of a real application server. In order to simulate a real Web e-commerce environment, we implemented the application server queuing model based on the proposal in [4]. It considers page size, number of images in each page, whether or not SSL is used, and whether cache is applied as four factors that determine server response time. The basis of the model creation is a case study of ShopIBM Web site. We ran tests based on our modelled environment and determined that the average response time of the application server should be 0.5 s. We changed the application service capability and found that when we set the number of concurrent requests the application server could service to 41, the response time is most close to 0.5 s. Therefore, we set the application server in our system model to be able to handle 41 concurrent user requests. All the upcoming requests from clients are first stored in a request queue and then serviced from there. We add time delay during the servlets execution to simulate the application server response factor.

3.2.6. Network connection. We made a simplifying assumption that the client side (proxy and individual Web browsers) and remote server are connected through a high-speed network, i.e., we did not simulate the data transfer exactly the same as it would occur in real world. Our simulation model is that for each data transfer, the network delay time is decided by a randomly generated number between 1 and 3 s.

3.3. System parameters

3.3.1. Hardware configuration. In our experiments, the server is a PC with Pentium III 1 GHz CPU, 512 MB RAM and 80 GB hard disk. We install Tomcat Web server, all servlets and image files for the bookstore, and application server on it. Client machine is a PC with Pentium III 800 MHz CPU, 128 MB RAM and 30 GB hard disk. It is used both as proxy server that runs RBE, and client machines because there are multiple emulated browsers generated and managed by the RBE. Both the server and client are on a department network with 100 MB data transfer speed. Both of them run on Windows 2000 operating system.

3.3.2. Database size (specified by TPC-W). As discussed earlier, database scaling is defined by the size (number of rows) of the `Item` table and the number of EBs configured

for WIPS, i.e., it is defined by the size of the store and size of the supported customer population. Our experiments use a database of 10,000 items. We did attempt to run the experiments with larger databases, but this was not possible due to limitations of the implementation that was used and the overhead imposed by Windows 2000 in searching large number of directories. According to the Java implementation of the benchmark, all the image files for book items are stored in file system instead of database. For each book item, it requires a regular image file of around 250 KB, and a thumbnail file of about 6 KB. 100,000 items require a file structure of 100 directories, each directory containing 2,000 files, and the total size of all the images is approximately 26 GB. We found that the time delay at searching image files in the file structure is significantly long.

3.3.3. The number of EBs (specified by TPC-W). The higher the number of EBs, the heavier is the workload that the Web server, the application server, and the network between server and proxy will experience. If the benchmark runs with more than 40 EBs, the capability of the application server will force some of the EBs to wait for some time before their requests could be served. If client-side caching algorithms are applied, since all the caches are implemented in main memory, the cache size for each EB is limited by the number of EBs due to the 128MB maximum client RAM space. We change the number of EBs at first to see the impact of user population on the system throughput under no-cache and infinite-cache scenario. While comparing the performance of different consistency algorithms, we fix the number of EBs to 100.

3.3.4. Web interaction mixes (specified by TPC-W). As indicated earlier the benchmark defines three distinct Web interaction mixes: browsing, shopping and ordering. Since we are more interested in a Web commerce environment where database update happens from time to time, we choose to focus on the ordering mix. In other words, no matter what the system parameters are, half of the Web user URL requests are browsing related and the other half are ordering related. The URL generation rules specified in the benchmark specification is strictly followed.

3.3.5. Cache size. Except for the infinite cache, we specify the maximum cache capacity in the program. This means that, when the benchmark has been running for a certain period of time, the caches will become full and LRU-SIZE cache replacement algorithm is used to remove the least recently accessed Web object(s) and make room for newly arrived ones.

For client-side cache, we start with 100 KB, and increase in increments of 100 KB up to a cache size of 800 KB. For proxy-side cache, since multiple EBs (simulated Web users) will share one single cache residing on the proxy, we set its initial capacity to be 50 MB, and increase it by 10 MB. The maximum cache size will be 100 MB (almost reaching the memory limit). This maximum value can be increased by increasing the available RAM on the testing machine.

3.3.6. Measurement interval. We started running the benchmark with an initial measurement interval of 1 hour. Later on, in order to find the performance bottleneck for each

algorithm (especially TTL), we increased the measurement interval by 1 hour each time. The maximum time duration of benchmark running is 6 hours. For the proxy-side infinite cache, a 3-hour benchmark running with 100 EBs (concurrent Web users) will increase the cache size up to 100 MB.

4. Performance results

We are interested not only in the relative performance of caching algorithms, but the location of the cache as well. For the same consistency algorithm, a browser cache will have different performance impact on the system compared to a proxy cache or server side cache. Conceptually, the contribution of server cache is limited due to its physical location, i.e., it is only capable of caching contents on one single server. Because of this, we ignore the possible deployment location of server cache in this research. Each of the examined algorithms are implemented both in a browser cache and a proxy cache, and the results are compared horizontally (per algorithm) and vertically (per location).

The performance tests that we conducted measure system performance with respect to a number of metrics: Web interactions-per-second (WIPS), response time, hit ratio, and traffic load. The first two of these are metrics specified by TPC-W, the latter two are those we have added for a deeper understanding of the system behavior. We present the performance results according to these metrics, which are defined in the appropriate sections.

For each set of experiments, we first get the boundary conditions, i.e., worst and best cases. To test for boundary cases, we tested the system with no cache and with infinite cache. The worst case is when there is no cache, which establishes the lower bound of the performance. The best case is when the benchmark runs with proxy-side infinite cache. We do not have infinite space to implement a real infinite cache, but as long as there is no cache object removal during the entire Web transaction period, the cache could be regarded as infinite. We did not implement client-side infinite cache because of the hardware limitation of our experiments.

Infinite caching is not a consistency algorithm. It does not enforce any cache consistency at all. All it does is to cache all the objects that EB ever requests. This will establish the upper bound of cache hit ratio and WIPS throughput. Obviously, infinite caching may not always give upper bound performance result. In some cases it might even perform worse than a finite cache or a system without cache. This is because as the cache size grows, it takes longer for the cache manager to search through the cache. If there are too many cached objects, it might be faster to simply request the object directly from origin server. There might be a certain cache size threshold for a specific Web environment. We do not consider this issue further in this research.

4.1. WIPS – System throughput

The TPC-W benchmark measures the number of successful Web Interactions Per Second (WIPS), given a particular workload and response time constraints. Each Web page is usually made of several components (e.g., text, forms, images, etc.), which, when put

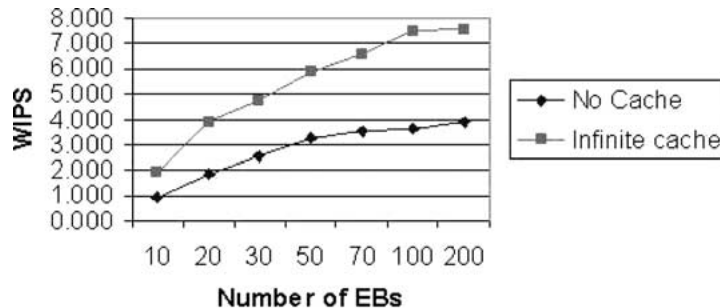


Figure 4. System throughput versus client population (boundary conditions).

together, form the entire Web page in the browser. Each of these components are unique objects and have to be retrieved separately from the Web server. A Web interaction is defined as the complete transfer of a Web page, including all objects, from the Web server to the user's browser. Basically the WIPS value should be stable in a small range given a fixed EB number, cache size and server response time, and it should increase with more EBs. But we expect a boundary value where the WIPS becomes stable even when the number of EBs increases dramatically.

As a starting point, we wanted to see the impact of client population on system throughput under the best and worst cases. We changed the number of EBs while fixing all other system parameters, and then ran the benchmark first without cache, then with infinite cache at proxy side. Figure 4 illustrates the change of WIPS as a result of change in client population.

With an infinite cache, regardless of client population, the system throughput is almost twice as high as when there is no cache. For each setting, it seems that the WIPS increases linearly as client population increases, but becomes stable as the number of clients reaches 100. This is due to the limited capacity of the system itself. According to the benchmark specification, under the ideal system configuration, the WIPS number should reach approximately the number of EBs divided by 7. In other words, a benchmark running with 100 EBs on an extremely fast system should get about 14.3 WIPS. This is far more than what was obtained in our experiments (for infinite cache with 100 EBs, our result is about 7.5 WIPS; without cache, WIPS is 3.99). The reason is that the ideal system configuration assumes an extremely fast Web server, no delay at application server side, and no delay at the network. This is not feasible in the real world. In fact, we simulated network delay by following a random distribution of seconds between 1 and 3. We also added an application server queuing manager to simulate the delay at application server side. The system environment we set up for our experiment is realistic and comparable to real-world Web behavior.

Locating the cache at either client side or proxy server side is expected to impact the WIPS differently. Therefore, we implemented TTL, Invalidation (INV), Polling-every-time (POL) and Lease algorithms on both proxy cache and browser cache. Figure 5 is the throughput comparison of client-side deployment, and Figure 6 shows the result of proxy-side deployment. As mentioned before, due to the main memory limitation, the client-side

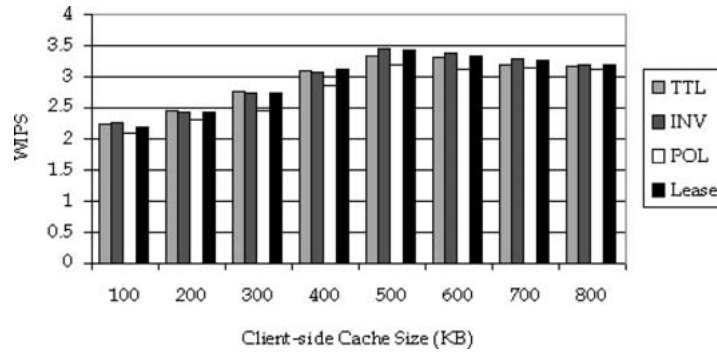


Figure 5. WIPS comparison of client-side algorithms (fixing 100 EBs).

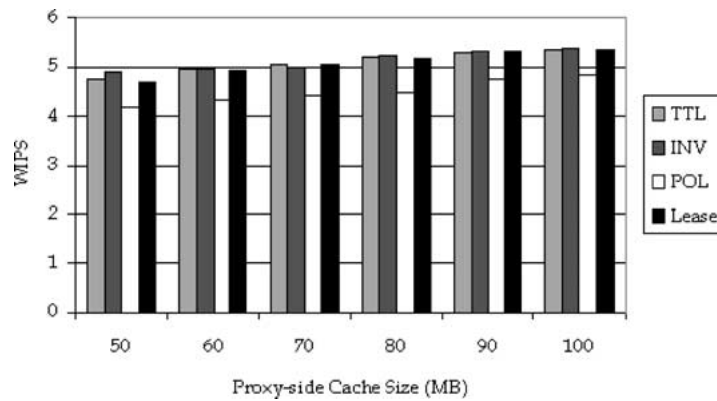


Figure 6. WIPS comparison of proxy-side algorithms (fixing 100 EBs).

cache size cannot exceed 1 MB. We started with 100 KB and increased it by 100 KB till 800 KB. As expected, the system throughput increases as a result of client population increase. Regardless of cache size, INV has similar throughput as Lease. When cache size is small, TTL beats Lease, but as the size grows, Lease algorithm outperforms TTL, and INV is always the best. The reason why Lease does not perform as well as INV is that sometimes a lease expires before the objects actually get updated, which results in extra validation delay. Expecting that a longer lease period might boost system throughput, we experimented with different lease periods. The result is that a longer lease period does improve system throughput, however, it is always lower than INV. This indicates the advantage of event-driven algorithms over time-based ones in an online Web commerce environment.

TTL, INV and Lease perform better than POL, although the difference is minor. One interesting finding is that for client-cache size, 500 KB seems to be the optimum for the system setup used in this study. As depicted in Figure 5, the WIPS decreases when cache size gets larger than 500 KB. This is because at a certain point when there are considerably

large number of objects in cache, the time delay due to cache search more than offsets the time saved by serving object from cache instead of origin server. In the real-world scenario, if the network connection speed is sufficiently fast, we do not recommend having a browser cache of very large size unless the cache manager has a very efficient hash table or other mechanism for object search.

From Figure 6 we could see that the increase of proxy-side cache size does not have as dramatic an effect on WIPS as client-side cache does. TTL, Lease and INV still perform very close while outperforming POL. This further proves that a strong consistency algorithm can do as well as a weak one. Notice that both TTL and Lease are time-based algorithms, while INV is completely event driven. The increase of cache size will result in higher cache hit rates for all algorithms, but for POL, it means more number of polling messages sent to server, which increases network traffic. In a network environment where connection speed is relatively slow, polling-every-time might perform even worse than having no cache at all.

While running the above two sets of experiments, we fixed 100 emulated browsers as the client population for the purpose of comparison. In the following experiments, unless explicitly specified, all results are obtained when the benchmark runs under the client population of 100 EBs.

The vertical comparison of consistency algorithms, i.e., their WIPS results for client-side and proxy-side cache, is illustrated in Figure 7. For comparison purposes, since the cache size for each of the 100 EBs is 800 KB in the case of browser cache, we chose the proxy-cache result when the cache size is 80 MB.

Obviously, with proxy-side cache, any of these consistency algorithms performs significantly better than with client-side cache. This is because proxy-side cache increases information sharing, which in turn increases system throughput. It is also likely that the limitation of our hardware configuration limits client-side caching performance significantly. Furthermore, for client-side cache, these algorithms do not have much difference with regard to WIPS, but for proxy-side cache, TTL, INV and Lease have similar throughput, and are all about 15% better than POL. The similarity of results at client-cache scenario is mainly because of the cache replacement time delay due to limited cache size, which constitutes a majority of factors that affect system throughput. For proxy-side cache, the cache

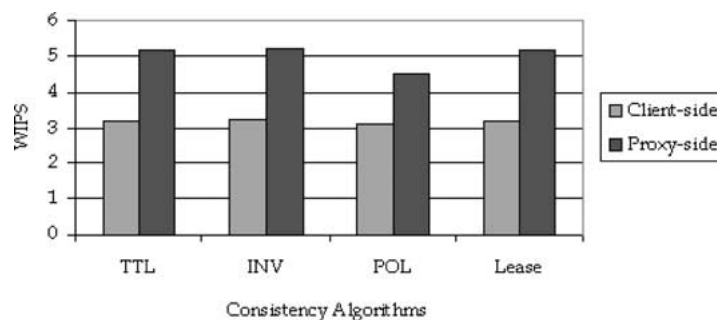


Figure 7. WIPS comparison of consistency algorithms at different locations.

size is bigger, therefore, the impact of polling messages is reflected more significantly in the results.

4.2. Response time

WIPS reflects the system capability of handling a large number of user requests. This is a factor that most online e-business competitors are concerned with. Meanwhile, response time, as another very important performance metric for e-business, affects directly the degree of customer satisfaction as it reflects the client latency. The famous folkloric *8-second rule* is a hard evidence. If a Web site takes more than 8 s to deliver a page, the visitors are very likely to leave the site.

TPC-W defines the response time of each successful Web interaction as the period between the time when the first byte of the first HTTP request of the Web interaction is sent by the EB to the SUT, to the time when the last byte of the last HTTP responses that completes the Web interaction is received by the EB from the SUT. Within the framework of this study, it is particularly important since it shows whether a cache consistency or replacement algorithm is efficient enough to increase the response time at the minimum, and whether the caching architecture would have adverse impact on the overall system performance.¹

In these experiments, the first objective is to see the comparison at boundary conditions. Fixing client population of 100 EBs, the benchmark is run under different measurement intervals, from 1 to 6 hours. The benchmark was run first without cache, and then with proxy-side infinite cache. Figure 8 shows the results.

Without cache, the response time remains almost unchanged no matter how long the measurement interval is. This is because whenever a user request is submitted, it has to experience the network and application server delay. By contrast, with infinite cache, the response time first drops, but then increases steadily although very little. Besides the factor of randomly generated network delay (same for non-cache scenario), this change in response time is mainly due to the time delay as a result of cache search. Again, having

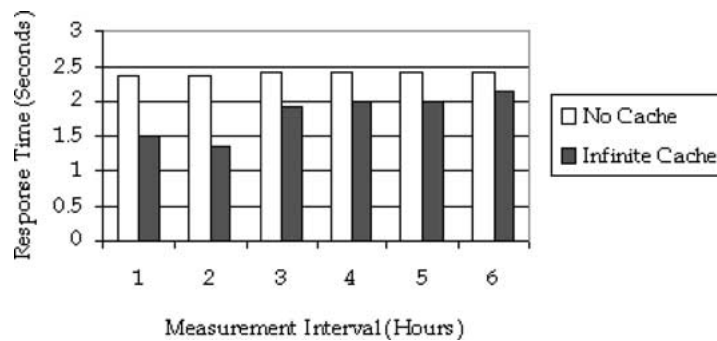


Figure 8. Response time under boundary conditions.

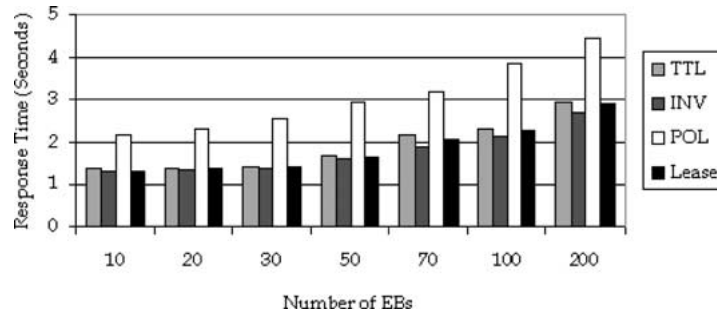


Figure 9. Response time using client-side consistency algorithms.

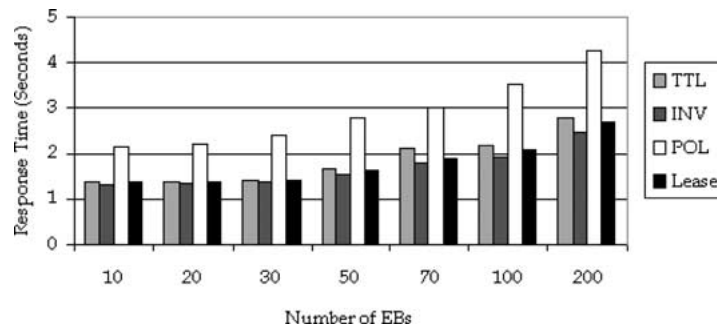


Figure 10. Response time using proxy-side consistency algorithms.

a large size cache is not always desirable, unless efficient cache search algorithms are implemented.

In order to find out the impact of client population on response time, as well as the cache deployment mechanism, the benchmark was run with client-side cache consistency algorithms, fixing cache size of each EB to be 500 KB. Figure 9 is the result of the experiments. Then the same set of experiments were run with proxy-side consistency algorithms, fixing proxy cache size to be 100 MB. The experiment results are displayed in Figure 10. Regardless of client population or cache location, INV performs a bit better than TTL and Lease with respect to response time. This suggests that an event-driven consistency algorithm might be more suitable for online e-business than time-based ones. On the other hand, POL is event-driven, too (because validation message is sent out only when a cache hit happens), but it results in the longest response time, and it becomes worse as the number of EBs increases. This is because, as the client population increases, the objects stored in proxy cache accumulate quickly and after a while it takes significantly longer to search an object in the cache, and even if the object is found in cache, the cache manager still needs to send out an IMS message to validate its freshness. Therefore, for that Web interaction, end user will experience not only cache search delay, but network and application server delay as well. Increased number of EBs result in longer queuing waiting time at application server side.

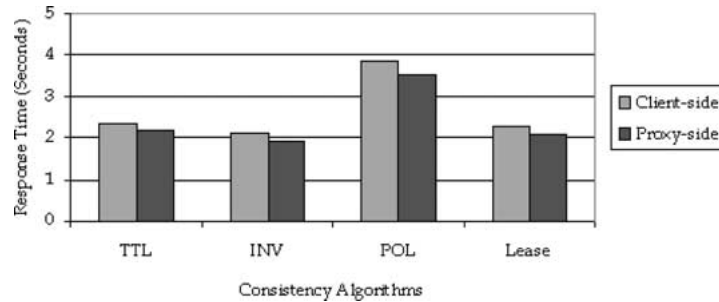


Figure 11. Response time under different cache locations.

The results of vertical comparison of the algorithms with respect to response time are depicted in Figure 11. POL is the least favorable algorithm regardless of cache location. Similar to system throughput experiments, if the network transfer speed is slow, POL is expected to perform even worse.

4.3. Hit ratio

Cache hit ratio is defined as the ratio of total number of bytes that all the EBs received from caches to the total number of bytes all the EBs actually received, either from cache or original server. The total number of bytes caches provide to end users is called *saved traffic*, and it is calculated by recording the size and number of hits on each cached object. A global count is kept in each cache to avoid miscalculation. This is mainly because when a cached object is evicted from cache by LRU-SIZE algorithm, all the information of this object is lost, such as its size and number of hits on the object.

Stale hit ratio is an important measurement factor for consistency control algorithms. It is measured by calculating the portion of stale cache hits over the total number of hits. For a fixed cache size, we want to find out whether this value gets bigger or smaller as measurement interval increases.

Hit ratio is a performance metric for cache deployment. It does not make sense to measure the hit ratio if there is no cache at all. Therefore in the boundary case, the benchmark is run only with proxy-side infinite cache. The measurement interval is first fixed at 1 hour, the client population is varied from 10 to 200 concurrent EBs, and the benchmark is run with a proxy cache that has 100 MB storage space. Up to 50 EBs there was no cache replacement during the 1-hour execution. However, when there are more than 50 EBs, the proxy cache soon fills up. We terminated the benchmark when cache manager realized that there was not enough cache space for the next object, and analyzed the Web interactions up to that moment. Figure 12 shows the cache hit ratio under different client populations. The result shows that with 100 EBs, we could achieve hit ratio of about 43%. If the cache space is big enough, this number is expected to be higher.

The measurement interval is then fixed to be 1 hour, the client population is varied, and the benchmark is ran with different consistency algorithms first with client-side cache,

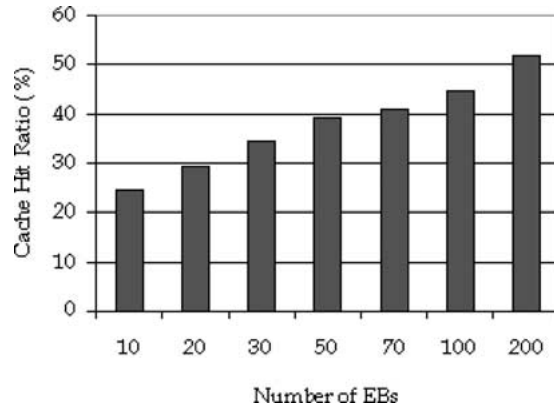


Figure 12. Cache hit ratio under proxy-side infinite cache.

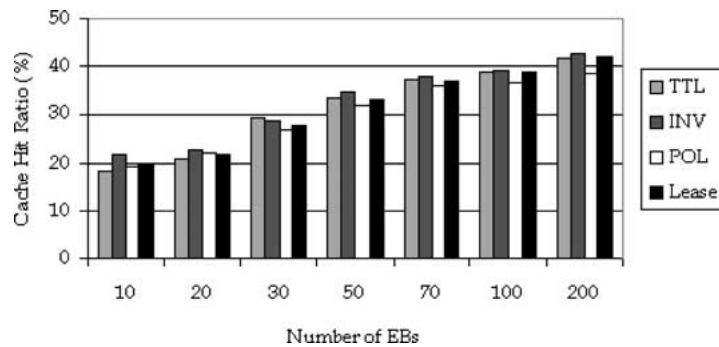


Figure 13. Comparison of client-side consistency algorithms.

then with proxy-side. Figure 13 shows the change of cache hit ratio under client-side cache deployment, and Figure 14 gives the result of proxy-side deployment. The results do not show much difference, because the cache hits include both valid and stale hits. We analyzed the data and separated stale hits from valid ones, Figure 16 displays the results.

We extracted from the above experiment data those that correspond to a client population of 100 EBs. The hit ratio in this population was compared under either client-side or proxy-side cache deployment. Figure 15 illustrates the results. Still, INV outperforms all other algorithms and achieves the highest cache hit ratio. Notice that for proxy-side cache, POL has almost the same cache hit ratio as TTL, while both of them achieve a little less than Lease algorithm does. The explanation is that we count both valid and stale hit as cache hit. Therefore, even though the TTL value has expired and the cached pages are stale, cache hit could still happen on these pages.

We are also interested in the stale hit ratio comparison under various consistency control algorithms. Figure 16 shows the stale hit ratio while applying TTL and POL both at client and proxy side. The stale hit ratio is not very high in each case, because an online bookstore

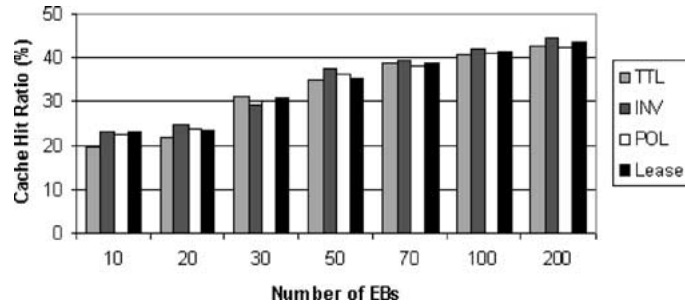


Figure 14. Comparison of proxy-side consistency algorithms.

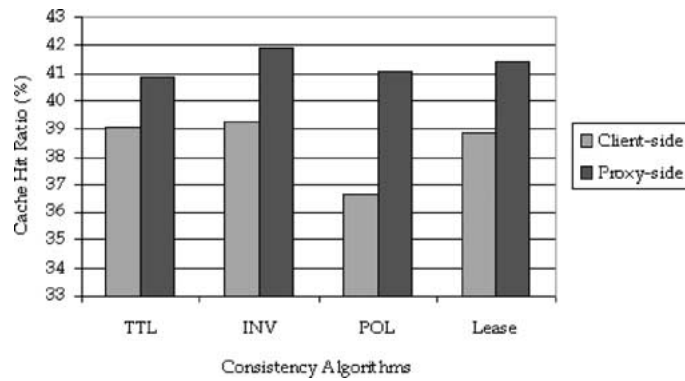


Figure 15. Cache hit ratio under different cache locations.

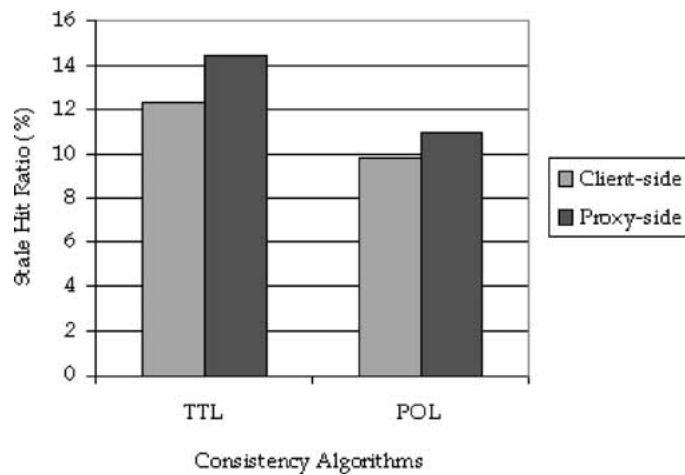


Figure 16. Stale hit ratio under different cache locations.

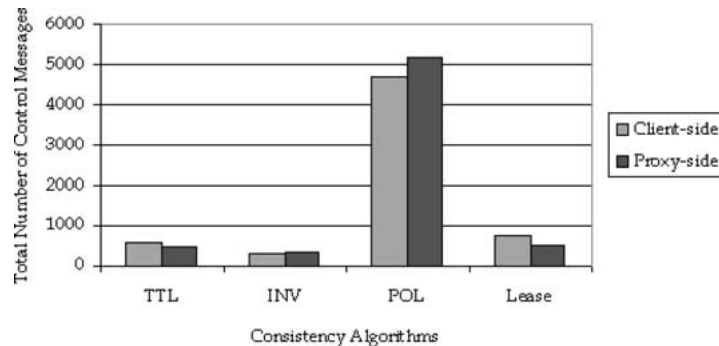


Figure 17. Control message comparison using different consistency algorithms.

is not a high-update application (compared to other transactional applications), therefore, most objects remain valid for quite a while. Meanwhile, even if an object is updated, it might not be requested at all. In other words, in the TPC-W scenario, cache hits do not often happen on stale objects.

4.4. Traffic

There are two types of traffic measurements in our experiments. The first is the total number and size of control messages. For TTL algorithm, there will be IMS and server acknowledgement messages. For invalidation and polling-every-time algorithms, there will be only one-directional control messages. This partly affects network traffic. The other type is the total size of all Web objects (HTML pages, images, etc.). This is the majority of online traffic. We explore them separately in our experiments.

In order to measure the impact of consistency algorithms on network traffic, we analyzed the control messages generated while running the benchmark. Since the size of the control messages, either IMS or acknowledgement notice, is in the order of bytes, we just counted the total number of messages for each algorithm. Figure 17 shows the total number of control messages under each algorithm, both at client side and proxy side. We see that POL generated a lot more control messages than other algorithms did. This creates extra delay experienced by end user.

We also compared the total number of bytes of objects that users receive and the total bytes of objects that are served by cache. For this experiment, the client population is fixed as 100 emulated browsers, the measurement interval is 1 hour, and the cache is at proxy-server side. Figure 18 gives the experiment results.

The same set of experiments were conducted with client-side cache. This time there is no boundary case since we did not implement client-side infinite cache as discussed earlier. Figure 19 gives the experiment results. There is not much difference on traffic savings for these algorithms; thus, with respect to network traffic saving, these algorithms perform similarly. Considering that INV and Lease provide strong consistency control, we would recommend either Lease or Invalidation algorithm as the better choice.

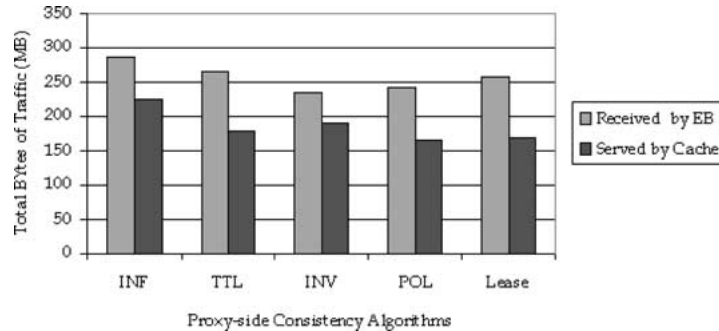


Figure 18. Traffic comparison under different proxy-side consistency algorithms.

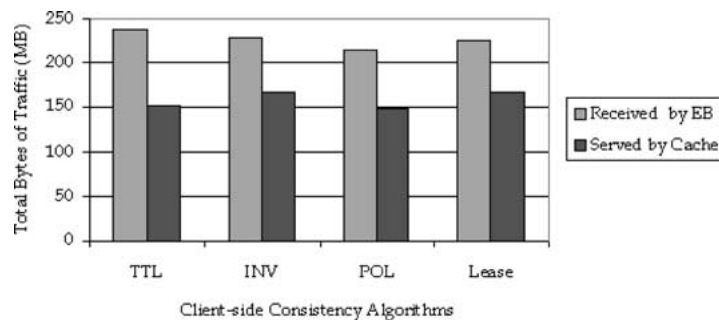


Figure 19. Traffic comparison under different client-side consistency algorithms.

We should point out that our experiment system is not in an isolated environment. Instead, it is part of the campus network whose data transfer speed is not stable from time to time due to ad hoc network activities. During our experiments, the network sometimes became extremely congested, which resulted in significantly lower system throughput and much longer response time. It is not reasonable that we include such results in the performance comparison and analysis. These outlier results are discarded in what is reported.

5. Conclusions

In this paper we study the performance characteristics of representative strong consistency Web caching algorithms using the TPC-W benchmark. To our knowledge, this is the first extensive performance study of these algorithms using an industry-standard methodology. The experimentation environment allows reasonable comparison of results.

Our results indicate that proxy side caching improves system performance more than client side caching and Invalidation algorithm performs best regardless of cache location. Based on our results, Polling-every-time is the least favorable alternative; it has the longest response time, producing the most message traffic. On the other hand, it is the easiest to implement.

The advantages of Invalidation algorithm over TTL and Polling-every-time are twofold. First, Invalidation has the fewest control message transfer, shortest response time, similar throughput as TTL while keeping cached objects fresh. Secondly, INV is event-based, which is more suitable for online e-commerce. The reason is that most Web data are not set to be changed at certain time; rather, Web objects get updated because of some event, e.g., client request. Event-based consistency control helps reduce unnecessary control messages while enforcing strong consistency more efficiently.

As expected, the performance of proxy-side cache is significantly superior to that of client-side cache. Having the cache shared by a group of users reduces redundant object storage. It is also easier to maintain consistency between server and one (or few) proxy caches rather than many browser caches. However, the hardware limitation of our experimental platform affects the client-side cache performance. The results suggest that browser cache is not as favorable as to proxy cache even when the storage space is not an issue.

Our results may not be optimum, partly because we chose the simplest and generally accepted LRU algorithm as our cache replacement algorithm. We did make some enhancements to the basic LRU, but as Cao and Irani point out, there are many factors that should be taken into consideration when designing cache replacement algorithm [1]. These factors include document download time delay, network travelling costs of different Web objects, etc. It is not yet clear what the impact of the cache replacement algorithm is on various caching algorithms. If the impact is uniform, then the relative results produced by our experiments are still valid. However, this is an issue for further study.

There are a number of directions that we are currently pursuing. One is the evaluation of various algorithms under a cooperative caching architecture. In our current experiment settings, all the EBs are managed by one single RBE. For large scale organizations who have thousands of end users possibly located at geographically separated sites, the use of a single proxy cache by all users will certainly not achieve desirable system performance. Running the benchmark under a cooperative caching architecture with multiple RBEs, each of which manages a group of EBs should give interesting insights.

Piggybacking, and thereby optimizing communication, is likely to produce savings. Other studies on caching systems within database environments (e.g., [9]) have suggested the advantages of this approach. It is likely that these advantages would hold in the case of Web caching as well.

Acknowledgements

This research was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) and a grant from the Canadian Institute for Telecommunications Research (CITR), a Network of Centre of Excellence funded by the Government of Canada.

Note

1. Certainly, WIPS can also measure these due to the well-known relationship between system throughput and response time.

References

- [1] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, 1997.
- [2] P. Cao and C. Liu, "Maintaining strong cache consistency in the World Wide Web," *IEEE Transactions on Computers* 47(4), 1998, 445–457.
- [3] V. Cate, "Alex – A global file system," in *Proceedings of the USENIX File System Workshop*, Ann Arbor, MI, 1992, pp. 1–11.
- [4] H. K. Edwards, M. A. Bauer, H. Lutfiyya, Y. Chan, M. Shields, and P. Woo, "A methodology and implementation for analytic modeling in electronic commerce applications," in *Proceedings of the International Symposium on Electronic Commerce*, 2001, pp. 148–157.
- [5] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989, pp. 202–210.
- [6] J. Gwertzman and M. Seltzer, "World Wide Web cache consistency," in *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, 1996, pp. 141–152.
- [7] B. Krishnamurthy and C. E. Wills, "Study of piggyback cache validation for proxy caches in the World Wide Web," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, 1997, pp. 1–12.
- [8] B. Krishnamurthy and C. E. Wills, "Piggyback Server Invalidation for Proxy Cache Coherency," *Computer Networks and ISDN Systems* 30(1–7), 1998, 185–193.
- [9] M. T. Özsu, K. Voruganti, and R. C. Unrau, "An asynchronous avoidance-based cache consistency algorithm for client caching DBMSs," in *VLDB'98, Proceedings of 24th International Conference on Very Large Data Bases*, A. Gupta, O. Shmueli, and J. Widom, Eds., August 24–27, 1998, New York City, NY, Morgan Kaufmann, 1998, pp. 440–451.
- [10] M. Rabinovich and O. Spatscheck, *Web Caching and Replication*, Addison-Wesley, 2002.
- [11] TPC, "TPC Benchmark™ W (Web commerce) specification version 1.4," Technical report, Transaction Processing Performance Council, 2001, <http://www.tpc.org/tpcw>
- [12] D. Wessels, "Intelligent caching for World-Wide Web objects," in *Proc. INET '95 Conference*, Honolulu, Hawaii, 1995.
- [13] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Using leases to support server-driven consistency in large-scale systems," in *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, 1998, pp. 285–294.
- [14] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Volume leases for consistency in large-scale systems," *Knowledge and Data Engineering* 11(4), 1999, 563–576.
- [15] H. Yu, L. Breslau, and S. Shenker, "A scalable Web cache consistency architecture," in *Proceedings of the ACM SIGCOMM'99*, Boston, MA, 1999, pp. 163–174.