The Evaluation of Strong Web Caching Consistency Algorithms

by

Yuxin Cao

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2002

Abstract

As the World Wide Web continues to grow in an exponential rate, Web Caching has become a hot research area, in the hope that by using it, we could not only reduce the client observed latency, but the network traffic and server load as well. Traditional wisdom holds that strong cache consistency is too expensive for the Web [CL98] because a lot of extra resource is required to enforce that. However, as business transactions on the Web become more popular, strong consistency will get widely accepted and required by popular online applications. This thesis evaluates the performance of different categories of cache consistency algorithms using TPC-W, the Web commerce benchmark. In order to decide on the optimum cache deployment location, we also conduct a number of experiments using the 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. Proxy-side cache has a 30-35% performance advantage over client-side cache with regard to system throughput.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to finish this thesis. I want to thank Dr. Kaladhar Voruganti of IBM Almaden Research Lab for his prompt responses and helpful suggestions at the early stage of my research work. I also want to thank Ivan Bowman who gave me precious suggestions when I set up the environment for the experiments. Appreciation also goes to Keith Edwards of University of Western Ontario for his valuable input on Application Server Queuing Model setup.

I am deeply indebted to my supervisor Professor Dr. M. Tamer Ozsu whose help, stimulating suggestions and encouragement helped me during the course of my research for and writing of this thesis. His hardworking spirit and persistent pursue of best solutions to research problems inspire me all the time.

My sincere thanks also go to Dr. Johnny Wong and Dr. David Toman, who took the time to read my thesis and gave valuable comments and feedback while they were very busy with their own work.

I want to give my special thanks to my parents for their support and encouragement to my study, work and life.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

As a large-scale interaction of human activity and computer systems, the World-Wide Web (in this thesis we use WWW or Web as reference to this term) has helped to push the modern world into a true information society. While the traditional business model does not have anything to do with the Internet at all, more and more e-Business Web sites have emerged in the past several years, representing the fierce competition of customer and business opportunities on the Web. Not surprisingly, there has been much recent interest in designing high performance electronic commerce Web sites to help maximize competitive advantage. These sites put heavy load on resources. Most of them could rely on *Web caching* to reduce network load, server load, and the latency of responses. Generally speaking, Web caching is a mechanism to store Web objects (such as HTML pages and image files) at certain location for convenient future access. It is a simple and easy way of providing fast response while reducing traffic jam on the Internet. However, while applying Web caching, *cache consistency* becomes 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, he/she would be accessing stale data. 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, researchers have made tremendous efforts to solve this issue by designing caching architectures and implementing various cache consistency algorithms. The main purpose of this thesis is to compare several representative consistency algorithms under the Web commerce environment, evaluate their performance using TPC-W [Tra01], 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.

1.2 Problem Definition

The size of the Web is increasing surprisingly fast. Researchers at the Online Computer Library Center (OCLC) claim that by October 2001, the Web contained some 8.4 million unique sites, compared to 7.1 million in year 2000. In 1999, this number was 4.66 million while in 1998 it was only about 2.6 million[1]. The rapid increase in Web usage has led to dramatically increased loads on the network infrastructure and on individual Web servers, as well as latency problems while accessing a

[1]http://www.pandia.com/sw-2001/57-websize.html

Web site. Exponential growth without scalability solutions will eventually result in prohibitive network load and unacceptable service response times. According to Zona Research, long Web page download times resulted in the loss of US$4.4 billion in e-Commerce revenue in 1999 alone[2]. As businesses turn more dependent on the Internet for additional customers and revenue, it becomes critical for these businesses to have high performing Web sites. Web caching emerged as an effective solution to these problems. In the context of WWW, caches act as intermediate service systems that intercept the end-users requests before they arrive at the remote server. Here is a general picture of how caching works. The cache manager checks to see if the requested object is available in its local storage, if it is, a reply is sent back to the user with the requested object; otherwise the cache forwards the request on behalf of the user to either another cache or to the origin server. When the cache manager receives the data, it keeps a copy in its local storage and forwards the object to the user. The copies kept in the cache are used for subsequent user requests. Finding (not finding) a copy in the local cache is referred to as a *cache hit* (*cache miss*).

In this thesis, unless explicitly addressed, the term *object* has the same meaning as *document* or *data*, if it refers to something that can be stored by Web cache.

A Web cache may be used within a Web browser that is installed on each physical PC/workstation, 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,

[2]http://www.clickarray.com/caching.htm

some classes of documents usually cannot be cached, such as scripts and pay-per-view documents. One focus in this thesis is client caching and unless otherwise noted, references to cache and cache manager should be interpreted as client cache and client cache manager (We will further explain the difference of the term 'client' and 'client-side' in Chapter 2).

Web caching has three main advantages [Wes95]: 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 [Wes95]. Together, these features can make the Web less expensive, less congested, and better performing.

Despite the above mentioned advantages, Web caching comes with some undesirable issues. One major issue 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. When the cache manager stores an object in its cache, how long can it be sure that this object will remain consistent with the original copy on the server? If the cache manager serves client with its cached object whose original copy has changed, the client gets a stale copy. Always keeping the cached copies up-to-date is possible if we keep contacting the server to validate their freshness, or if the server sends updates to cache each time the object is changed, which seems to be the only way to guarantee that a cached document is consistent with the version on the original server. 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.

1.3 Why Strong Cache Consistency Is Important

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 [GS96].

For many Internet services, strong consistency is not required, although it may be desired. One good example is online newspaper and journals. If it is a daily or weekly newspaper, the cache manager can just cache the documents and need not check for their freshness during the remaining time period (this is also an example of TTL cache consistency algorithm that we will discuss later). Even for those newspapers that do get updated frequently, we could still apply weak consistency because the worst case is the Web user reads out-of-date news, but nobody gets hurt.

In the e-business world, however, weak cache consistency might not only be unsatisfactory and even annoying, sometimes it will be completely unacceptable. Here are several example applications where strong cache consistency must be maintained.

Online Bookstore Today, like many other businesses, bookstores are extended to the Internet. Good examples of online bookstores include Amazon, Chapters, etc. Users can browse best-selling books, read customer reviews, compare prices, place their orders and buy books online. Let's say that a customer (could be an individual or business organization) wants to buy 3 copies of a book, he/she checks the available amount from Web page and it shows that the book still has 5 copies available, the customer might happily make 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 it is impossible for the customer to make the purchase at the time; in other words, the transaction is invalid, or at least won't guarantee that the user actually bought the book. The credit card transaction would have to be reversed, and the customer might rather go to a bookstore nearby and buy the books there. If this happens often, the Web users will be disappointed with the information they retrieve from the Web application, and lose interest in buying books online, because they got the feeling that what they see from the Web pages is not exactly the information at the moment. This could mean lost business to online stores.

Stock Quote Stock market is always a hot place to go. The up-and-down of index figures and prices of individual stocks often attract the attention of hundreds of millions of people. Since not very long time ago, people could get stock quotes online via the services provided by online brokerage companies. You enter the ticker for a stock, hit 'Enter' and you will see its most recent price and all the news and headlines related to it. Many people make buy/sell decisions based on the information they receive from the Internet. If there is a cache that services user requests from its cached copies, it is possible that such information is out-of-date. It is hard to imagine that a user would be happy if he/she made his/her stock-buying or stock-selling decision based on out-of-date information. 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 disastrous.

Online Auction This is another good example where strong cache consistency must be applied if Web caching mechanism is used. Online auction stores, such as *Ebay*, *Yahoo! Auction*, etc., auction various kinds of merchandise. 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 placed 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 because the bid might be too low due to the fact that it is based on outdated information. Weak cache consistency is completely intolerable here.

From the above examples (there are a lot more of them) we could see that weak cache consistency, although saving response time and network bandwidth, is not generally acceptable in electronic commerce, where information is sensitively related to time. On the contrary, strong cache consistency can well satisfy user's data freshness requirements, although it costs more resources.

1.4 Thesis Scope

As discussed in previous sections, there are both desirable features and undesirable drawbacks of applying either weak or strong cache consistency. This thesis evaluates 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 thesis, 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.

1.5 Thesis Organization

This thesis is organized as follows. The next chapter provides a comprehensive survey of state-of-the-art in Web caching. Chapter 3 introduces TPC-W benchmark, its system requirements, performance metrics and workload specification. The detailed description of the algorithms we implemented is given in Chapter 4. We discuss our simulation model and environment setup in Chapter 5. Chapter 6 gives the experimental results. We draw our conclusions and clarify possible future work in Chapter 7.

Chapter 2

Background and Related Work

The purpose of this chapter is to conduct a comprehensive background study in the literature of Web caching, identifying related work that has been done by other researchers.

2.1 Cache Deployment Strategies

As previously discussed, a cache could be deployed either in a browser, on proxy server, or at the Web server. There has been significant research on combinations of physical deployment strategies, cache consistency and/or replacement algorithms in order to achieve optimal performance gain in a Web environment. This section identifies major cache deployment strategies that are either widely used or proposed in the literature.

2.1.1 Browser Caching

Browser caching is the simplest form of Web caching. Most commercial Web browsers have caching facilities. The browser simply stores in local storage every Web object that the user accesses, and if the user requests the same object at a later time, the application gets the object from local cache storage instead of sending a request to the server. As most users have undoubtedly experienced, accessing a Web page that was visited just a moment ago could be very fast by simply pressing the "Back" button in the browser window. However, sometimes this could be annoying if the user wants to read the most recent version of the page. He/she might have to press "Reload" button to force the browser to fetch the Web page from origin server. On the other hand, if for a period of time the user keeps requesting new objects that he/she never requested before, the existence of a browser cache will help nothing but increase latency. The reason is that no matter whether a cache hit happens, the local cache manager searches cache upon each user request. The more objects the cache stores, the longer it takes the cache manager to finish its search. If the user requests a new object every time, every search results in a cache miss that takes more time, therefore client latency would be longer.

2.1.2 Server Caching

In order to reduce the workload of Web server (the original content provider), a server-side cache could also be deployed. The main purpose of server-side caching is to store documents, images and other frequently inquired database information in cache so that the need for redundant computation or database retrieval is reduced and server could be relieved from repeated requests. Needless to say, if we can cache database query results, a lot of database operation time delay could be saved. As Challenger et al. claim, the 1998 Olympic Winter Games Web site had cache hit rates close to 100% by placing caches in front of the Web server to instantly cache and update dynamically generated pages [CDI99]. Even without using their *Data Update Propagation* (DUP) algorithm, their system had cache hit rates of 80% [CDI99]. However, common sense suggests that in order to achieve 100% cache hit ratio, all the objects that can possibly be requested should be stored in cache. In other words, we need a cache storage no smaller than that of the content server. This might be feasible for an Olympic Games Web site, but is not a practical solution for most systems.

2.1.3 Proxy Caching

Proxy caching is probably the most popular and widely accepted general caching architecture. As its name indicates, proxy caches are usually deployed at the edges of a network (i.e., at company or institutional gateway/firewall hosts) where a proxy server resides, so that they can serve a large number of internal users. This can consequently reduce the bandwidth consumption on the wide area network, because a noticeable portion of the traffic is localized. In this thesis, in order to differentiate proxy caching from browser caching as two different choices of cache deployment location, we refer to browser caching as “client-side caching” and proxy caching as “proxy-side caching”, while they are both client caching compared to server caching.

One advantage of proxy-side caching over browser caching is that proxy caching improves information sharing. Let's assume that user A and user B are both in the same network and their browser caches have not cached any object yet. User A requests objects OBJ1, OBJ2, OBJ3 and OBJ4 respectively. Later, user B submits the same request for these four objects in the same sequence. With only local browser cache, both user A and B will experience zero cache hit rate. With a proxy cache in the network, however, all the requests from user B will be satisfied from the cache because those objects were already requested, retrieved and cached. In this simple example, the cache hit rate reaches 50%. The advantage of proxy caching, therefore, is that once one user requests a document which causes the proxy to fetch it from the Web server, the proxy keeps one copy for future sharing by all users within that network. The more users a proxy cache takes care of, the higher the possibility that a cached object will be requested again, either by the same user or some other users within the same network. Abrams et al. claim that a proxy cache that takes their workload has a 30-50% maximum possible hit rate regardless of the way it is designed [ASA+95], which is very attractive to network managers who want to reduce unnecessary network traffic as much as possible. This idea has been repeatedly visited in the area of dynamic content caching. We will discuss it in more detail in Section 2.1.6.

The example given above might not always be true. The reason is that proxy cache stores Web pages, many of which contain user-specific information, e.g., user id, browser id, shopping id, etc. If a Web page P1 is cached and it contains information about user A, then even if user B issues exactly the same request as user A, the cache manager is unable to feed B with page P1, just because of the user-specific part of the page. In order to be more flexible, and better utilize the Web pages that proxy cache stores, Luo and Naughton propose a framework called "Form-Based proxy caching" [LN01], in which the proxy cache manager could extract general information from a Web page and partly satisfy another user's request without submitting the whole request that results in significant database operation at the Web server side. Again, this falls into the category of dynamic content caching and we shall elaborate on it later.

The types of traffic that a browser, a proxy, and a server cache must manage are different. A browser cache responds to exactly one client, while a caching proxy server responds to a group of clients that have some relationship (e.g., members of the same domain or organization). Certain degree of similarity in browsing behavior is expected from those clients. Finally, a server cache responds to world-wide user requests, but stores only a limited set of objects that belong to the server on which it resides. Depending on the specific purpose of caching, any one of them could be the best choice.

In [KLM97], the authors further classify proxy caching into two categories: passive and active. *Passive caching* is one that only caches objects that have been requested by the user, while *active caching* has the ability to request objects before they are requested by the user. This concept has another name: *prefetching*. There is a whole class of prefetching algorithms. There is also a lot of debate on their pros and cons. We shall discuss the issues deeper in Section 2.3.4.

Besides cache hit ratio, another major performance indicator used widely to evaluate caching performance is *latency*, or *response time*. These two terms have the same meaning throughout this thesis. In most situations, a proxy is located much closer to the client than the Web server. Usually (and realistically) the latency between proxy and server contributes a large portion of the overall latency between the Web server and ultimate user. This part of the latency is called *external* latency, and the part between proxy and client is *internal* [KLM97]. A maximum of 26% latency could be reduced by using proxy caching that has no prefetching mechanism [KLM97]. With prefetching, the maximum latency reduction reaches 57%. However, Douglis et al. [CDF$^+$98] cast doubts on these latency reduction claims. They claim that the 26% maximum latency reduction is based on some simplifying assumptions such as ignoring requests that did not result in a successful retrieval. These unsuccessful requests that were ignored in [KLM97], on the contrary, contribute significantly to the total bandwidth requirement [CDF$^+$98], and consequently, the overall system performance.

Douglis et al. also argue that the interactions between HTTP, TCP, and the network environment should not be neglected when judging the performance impact of proxy caches [CDF$^+$98]. Due to the *slow start* feature of the protocol, a TCP connection takes a fair amount of time to reach full throughput [Luo98]. Since the cost of setting up TCP connection is high, and connection aborts between client and server (sometimes) more than offset the latency reduction that is achieved by proxy caches, Douglis et al. suggest a non-traditional role of the proxy as a *connection cache*, which caches connections in addition to data [CDF$^+$98]. By caching the connection between the end user and proxy, as well as the connection between proxy and the original server, there are several flexibilities. First, the connection cache needs only one (or small number of) persistent connection(s) between the proxy and the original server, regardless of the number of clients connected to the proxy. Second, each client needs to maintain only a small number of persistent connections to the proxy regardless of the number of servers it visits [CDF+98]. Finally, if for some reason the connection between the client and proxy or between proxy and the original server is broken down, significant performance benefits could still be achieved thanks to the connection cached on the other side.

Figure 2.1 is a general model of proxy caching. In order to implement a fully functioning Web proxy cache, a cache architecture requires several components such as a storage mechanism for storing the cache data, a mapping mechanism to establish relationship between the URLs and their respective cached copies, and the format of the cached object content and its metadata [Luo98].

As we mentioned, the general notion of deploying a proxy cache is to have it reside at network gateway or connection point between the organization and the Internet, so that the cache could serve the content from the Internet to a group of users. There is another way of deploying proxy cache, which is called *reverse proxy caching*, in which the proxy caches serve a specified group of Web servers to general Internet users [LN01]. This kind of deployment can also alleviate Web server workload while putting the cache closer to server instead of end user. This is a feasible solution to improving overall performance, but the performance gain is limited compared to having the cache on proxy server. So far, there is little

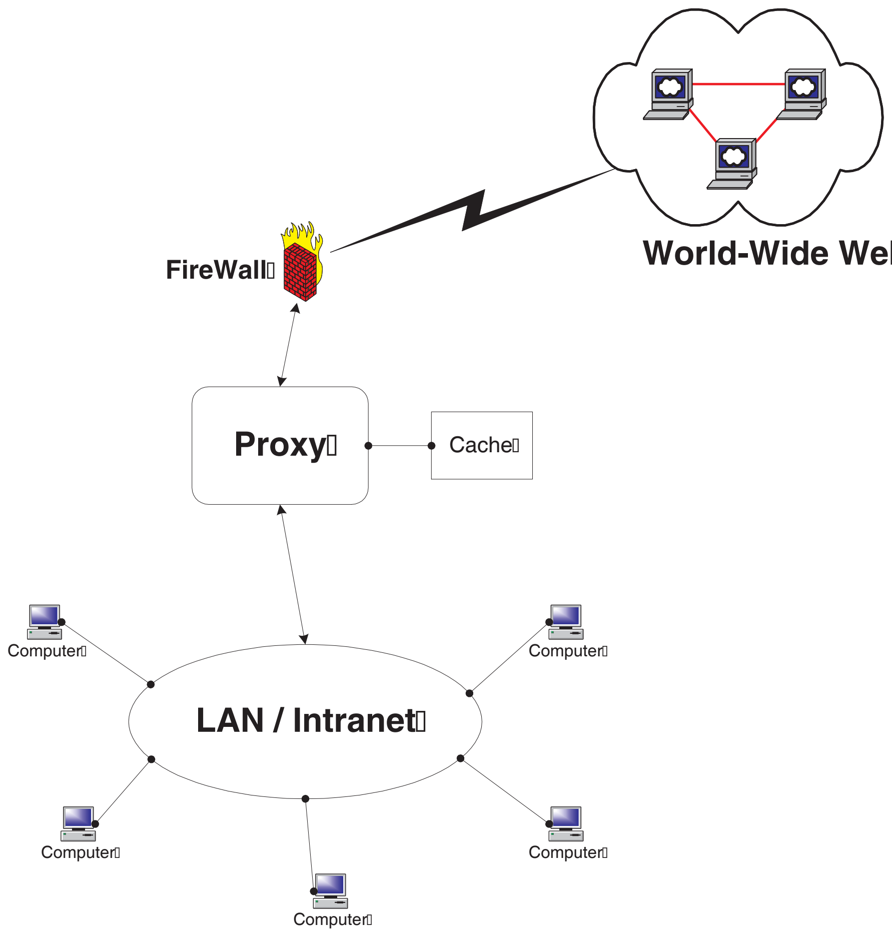

Figure 2.1: General model of Proxy Caching

published information available on reverse proxy caching, probably due to its own limitations.

2.1.4 Hierarchical Caching

Proxy caching is a general caching architecture based on the idea of maximizing the sharing of cached objects. In the Web environment, there are thousands of proxy caches and each stores objects that others might be requesting. Sharing information locally results in network bandwidth savings and server workload relief. Therefore, designing cooperative caching architectures have become attractive. Danzig et al. showed in their 1993 study of file transfer traffic on the National Science Foundation Network (NSFNET) that by deploying caches in a hierarchical architecture, the network bandwidth consumption could be greatly reduced [DHS93].

Let us first discuss how hierarchical caching works. Rodriguez et al. make a reasonable assumption that the Internet hierarchy consists of three tiers of ISPs: institutional networks, regional networks, and national backbones [RSB99]. Figure 2.2 displays the network topology of this model.

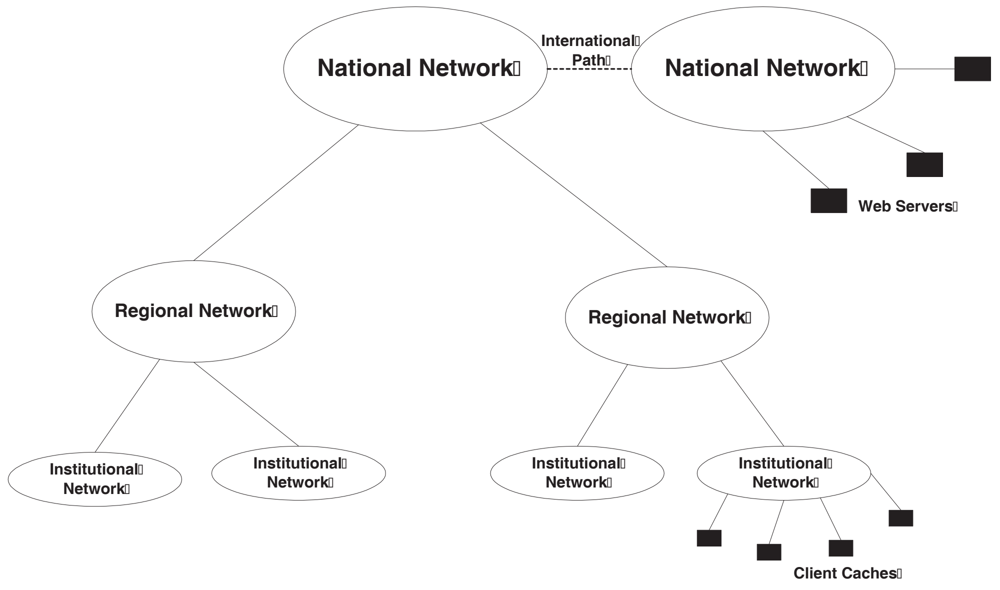

Figure 2.2: Network Topology (adapted from [RSB99])

At the bottom level of the hierarchy there are the *client* caches, which are also referred to as *leaf* caches [PH97]. One or multiple Web end users are attached to each client/leaf cache. In the situation of multiple end users, a client cache acts as proxy cache whose content is shared by all the users under it. When a request is not satisfied by the client cache, or in other words, a cache miss occurs, the request is forwarded to the *institutional* cache, which is one level above leaf cache. If the document is not found at the institutional level, the request is then forwarded to a higher level called *regional* cache. Regional cache further forwards unsatisfiable request to the *national* cache. If the requested object cannot be found at any cache level, the national cache contacts directly the original server, which results in file transfer through the Internet. When the object is found, either at a cache or at the Web server, it travels down the hierarchy, leaving a copy at each of the intermediate caches. Further requests for the same object travel up the caching hierarchy until it is found at a certain level. The term *siblings* refers to neighbor caches that are on the same level of the hierarchy. When a cache miss happens, the cache manager could also forward requests to its sibling caches and if any of them has the object, it would take less time retrieving the object compared to forwarding the request to the next higher level.

Hierarchical Web caching was first proposed in the Harvest project [DHS93, CDN+96]. By having a hierarchical arrangement of caches, wide-area network bandwidth demand could be reduced because a lot of information transfer is limited to within institutional or regional network, and redundant transfer of information at successive levels in the network is reduced to the minimum possible. Compared to single proxy caching, this is an improvement since it reduces traffic through an organization's gateway, as well as all gateways up to the root node of the hierarchy.

However, using hierarchical caching introduces undesirable features. First, since every object travelling down the hierarchy will leave a copy at each level, extra disk space may be required at each node of upper levels to store objects that are stored in nodes beneath it. For a node to maintain an optimal hit rate, and provide the maximum reduction in redundant transfers, its local cache should contain the union of the contents of all its child nodes. Obviously, the higher a node is in the hierarchy, the more storage space it needs. But the reality is that the number of end users on the Internet (who are all attached to leaf caches) increases exponentially, so it is impossible for each node to store all the information its child nodes request.

Second, for each user request, every level on the hierarchy introduces extra time delay, which affects overall system performance. If an object is not stored at any leaf cache, a user request for it will travel up the hierarchy until it hits the root node. The hit/miss check at each node level might be slow because the list of objects tends to grow bigger and bigger as the level goes higher. Chankhunthod et al. [CDN$^+$96] claim that if the hierarchy contains no more than 3 levels, it will add little noticeable access latency. However, a normal hierarchy, as shown in Figure 2.2, should have at least 4 levels of caches. After all, for a hierarchy as deep as only three levels of caches, the time delay issue still exists and cannot be eliminated. Their claim is not convincing. Our experiment results suggest that even when there is only one level of cache between the origin server and end user, the access delay at cache is not negligible, especially when the cache storage is rather big.

The nature of hierarchical caching is such that higher level caches will have to handle more requests coming from their child nodes. They will easily become bottlenecks of the system and result in long queuing delays that makes the latency issue even worse.

As previously discussed, in most hierarchical caching systems, missed cache requests are sent to sibling caches as well as upper level caches in the hierarchy so as to reduce object retrieval latency. This also creates maintenance problems because each node in the hierarchy has to maintain location information about all its siblings. While it is trivial to maintain such information when the hierarchy is within a single organization or local-area network, the task becomes increasingly complicated as the responsibility for maintenance is allocated throughout various organizations within a caching network. The dynamic nature of the Internet makes it more difficult than expected, considering the possibility of frequent cache addition and deletion.

Finally, from the cache consistency point of view, having multiple copies of the same object at different cache levels will potentially increase stale hit ratio and consistency maintenance costs if object update happens frequently.

2.1.5 Distributed Caching

Generally speaking, the benefits of applying hierarchical caching are somewhat offset by its disadvantages. As a result, researchers have proposed various schemes to better utilize cache resources to address the above issues. Most of these schemes belong to the category of **Distributed Caching**. In a distributed caching system, the hierarchical structure remains intact, but only leaf caches will actually store objects, and file transfers only happen at leaf cache level. Upper level caches would be used to maintain meta-data of objects stored in leaf caches, i.e., what object is stored in which cache. This scheme takes advantage of the hierarchy for quickly locating requested documents. At the same time it removes the requirement for upper level nodes to maintain large storage space.

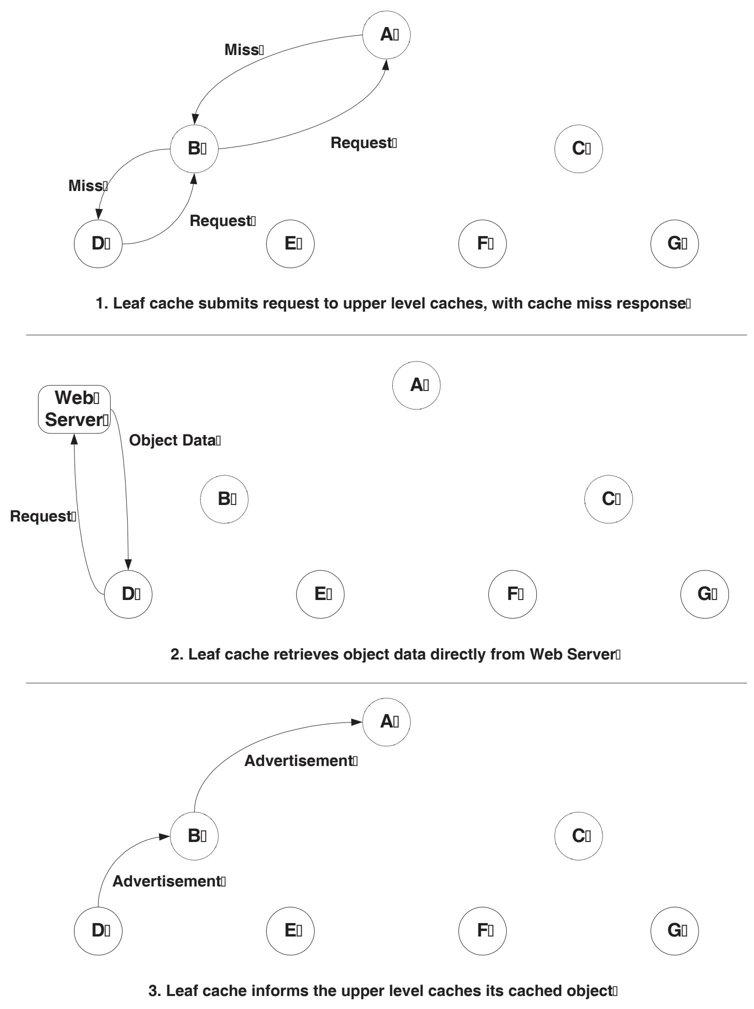
Miss
A
B
Request
C
Miss
Request
D
E
F
G
1. Leaf cache submits request to upper level caches, with cache miss response
Web Server
A
Object Data
B
C
Request
D
E
F
G
2. Leaf cache retrieves object data directly from Web Server
A
Advertisement
B
C
Advertisement
D
E
F
G
3. Leaf cache informs the upper level caches its cached object

Figure 2.3: A distributed caching scheme (a) (adapted from [PH97])

We borrow the figures in [PH97] to illustrate how a distributed caching system

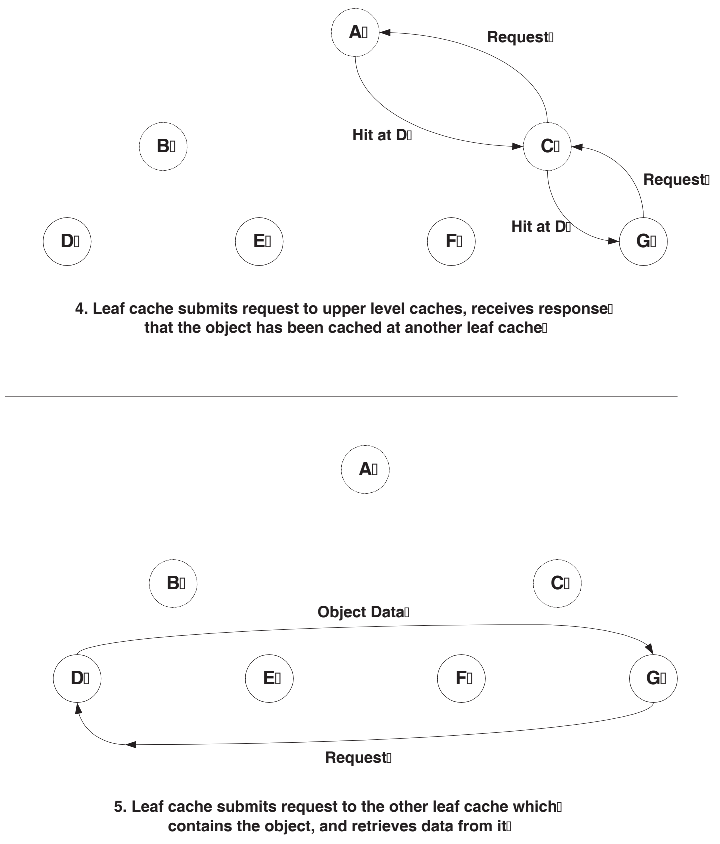
A
Request
B
Hit at D
C
Request
D
E
F
Hit at D
G
4. Leaf cache submits request to upper level caches, receives response that the object has been cached at another leaf cache
A
B
C
Object Data
D
E
F
G
Request
5. Leaf cache submits request to the other leaf cache which contains the object, and retrieves data from it

Figure 2.4: A distributed caching scheme (b) (adapted from [PH97])

generally works. As shown in Figure 2.3, D, E, F, G are leaf caches that cache objects. A, B and C are upper level caches that only maintain location information about objects. When a leaf cache D receives a user request for a document that it does not have, it forwards the request to upper levels until it reaches the root level cache and the "cache-miss" response gets all the way down to D. So cache D requests the document directly from the original server (notice that in this scheme, it is not the root cache but leaf cache who contacts origin server for object retrieval). After storing the object, D notifies its parent caches by "advertising". The advertisement message gets forwarded up to the root level cache A. Figure 2.4 explains what happens if later on another end user under leaf cache G in the network requests the same object. Since G does not contain the document, it forwards the request to upper level cache C, which in turn forwards the request to its parent cache A. Now that A has been informed that a copy of the object is present in leaf cache D, it sends the location information back down to leaf cache G, which results in direct object transfer from leaf cache D to G.

In [PH97], the authors exclude the possibility that each leaf cache could query its siblings to determine whether they store requested objects. Instead, each cache could only communicate with its parent or child caches. This rule was soon improved to allow more cooperation between sibling caches [CDN$^+$96, TDVK99, FCAB00, SMK$^+$01].

Cooperative caches usually need reliable cache-to-cache communication protocols for efficient object discovery and delivery. In the context of distributed caching, the Harvest group also designed the Internet Cache Protocol (ICP) [WC97b, WC97a], which supports document discovery and retrieval from sibling caches as well as parent caches. *Cache Digests* is proposed by Rousskov and Wessels [RW98] as an alternative to ICP. Under this scheme, each cache contains a digest (i.e., metadata) about the objects its neighbors have. Digests are made available via HTTP, and a cache downloads its neighbors' digests at the time it starts up. Another approach to distributed caching is the Cache Array Routing Protocol (CARP)[VR98], which has been applied in Microsoft Proxy Server 2.0. CARP uses hash-based routing to provide a deterministic "request resolution path" through an array of proxy caches. Each of these proxy caches has an array member identity. The request resolution path, based upon a hashing algorithm of the proxy array member identities and uniform resource locators (URLs), means that for any given URL request, the browser or downstream proxy server will know exactly where in the proxy cache array the information will be stored - whether already cached from a previous request, or making a first Internet hit for delivery and caching. Both Cache Digests and CARP have been tested and deployed in Squid[1] proxy caching system.

Distributed caching has become a hot research topic. Povey and Harrison proposed a distributed Internet cache [PH97]. In their scheme, upper level caches are replaced by directory servers which contain location hints about the documents stored at every cache. A hierarchical metadata-hierarchy is used to make the distribution of these location hints more efficient and scalable. Tewari et al. studied several Internet caching strategies and derived several basic design principles for distributed caching, such as sharing data among as many caches as possible, and caching data close to clients [TDVK99]. The authors proposed a similar approach by implementing a fully distributed Internet cache where location hints are replicated locally at the institutional caches [TDVK99], and *push-based* algorithms are used to dynamically push objects towards users who are more likely to request them. Zhang et al. propose a similar approach called *adaptive caching*, in which all distributed Web servers and cache servers are organized into co-related *multicast groups* [ZFJ97]. By using multicast, an object request will be satisfied by the nearest cache who has it. Popular Web pages travel through cache groups to get

[1]http://www.squid-cache.org/

closer to the caches who request them the most. One fatal flaw of this scheme, as pointed out by the authors [ZFJ97], is the limitation on scalability, because it is impossible to multicast all Web requests on the Internet.

So far we have discussed hierarchical and distributed caching architectures. Although the relative merits and performance characteristics of caching architectures is still a debatable issue, what is certain is that, as the size of the Internet grows, Web caching systems tend to become more cooperative, taking advantage of information sharing to save bandwidth consumption and reduce user perceived document retrieval latency. In [RSB99], the authors analyze the performance of hierarchical and distributed caching, as well as a hybrid scheme where caches cooperate at every network level of a caching hierarchy. Their experiments show that hierarchical caching has shorter connection times than distributed caching. Placing additional copies at intermediate levels reduces latency for small documents. Meanwhile, the results show that distributed caching has shorter transmission times and higher bandwidth usage than hierarchical caching. A “well configured” hybrid caching scheme can combine the advantages of both hierarchical and distributed caching, reducing the connection time as well as object transfer time.

2.1.6 Dynamic Content Caching

Despite various forms of caching strategies, proxy caching is still the most effective and well accepted mechanism to improve Web performance [YFIV00]. The cache hit rates of most proxy caches, however, are not always satisfying. By using traces, it has been shown that proxy caches can manage to obtain a maximum cache hit rate of about 50% [WVS+99], i.e., every one out of two user requests can not be serviced from the cache, and this is already the best case. This limitation is mainly due to the dynamic nature of many HTML documents that are generated based on user profiles and request parameters. As more and more Web sites evolve to provide e-commerce and personalized services to a wide range of application users, dynamic content generation becomes more popular, and caching dynamic contents becomes an increasingly important issue that affects the scalability of the Web.

Dynamic documents are typically generated using CGI scripts, Java Server Pages (JSP) or they include the result of a query to a database. Some Web pages might contain client-specific information as well, such as cookies. Measurement results show that 30% of user requests contain cookies [CDF+98, FCD+99] that make the document non-cacheable. In this thesis we are not concered with Web documents that contain cookies.

A Web page might contain both static content part that is always the same regardless of user request, and dynamic content part that is generated per user-specific request. If a cache manager caches such a Web page as a single and non-separable object, the next hit on this object is unlikely to happen unless the request is from the same user with exactly the same specific request parameters. In other words, many such objects would be useless although they are cached, which significantly reduces caching efficiency.

Researchers have come up with various schemes for caching systems to better utilize Web pages that contain dynamic contents. Cao et al. propose *active cache* [CZB98], in which a Java applet is attached to each URL or a collection of URLs.

When caching an object, the proxy also fetches the corresponding cache applet. If later on a request is received for that object, the applet is invoked with the user request and other information as parameters. The applet then decides whether it should return the cached object as is, rewrite the cached object, or direct the proxy cache manager to forward the user request to the origin server. The approach is very flexible and can be used to maintain consistency in an application-specific manner. Meanwhile, it is also capable of dynamically modifying existing documents without adding workload to the server. Despite the flexibility, active cache has its design limitations. For every user request, a new Java process needs to be started, which means extra memory space needs to be allocated for each request, even if the request is the same as a previous or simultaneous one. From security and robustness point of view, this design leaves the door open to denial-of-service attacks. A malicious Web user or group of users could deliberately send large number of requests at the same time, which results in server crash.

Smith et al. propose *Dynamic Content Caching Protocol* (DCCP) [SAYZ99], to allow individual content-generating applications to exploit query semantics and specify how their results should be cached and/or delivered. Using DCCP protocol, a Web application can specify result equivalence between different documents it generates. Proxy and other caching agents can make use of this information to speed up dynamic Web content delivery. However, DCCP has its design flaws as well, in that it has been designed to target only at GET-based queries, which have user specific parameters appending with '?' to the URL paths. However, there are a large number of Web requests that use POST-based queries which have their parameters sent in a separate line. For those queries, DCCP is useless.

We can regard Web pages that are changed or updated frequently as dynamic content as well. If a page has been changed on origin server and the cache still keeps the old version, there is a consistency issue that the system must resolve as early as possible to avoid sending stale copy of the page to user. Banga et al. suggest that only the differences between version of a page, or *delta*, be sent to cache manager so that the entire page does not need to be transferred every time the page is changed, considering the change might be minor [BDR97]. This solution is mainly to deal with latency issue. Of course, it can also save bandwidth because fewer bytes will be transfered, assuming the delta part is smaller than the Web page itself. However, since the delta-encoding is based on at least one basic form of the Web page, cache manager will have to store a potentially large number of past versions. Meanwhile, server and each cache manager need to have at least one common base version of the page to compute delta upon, not to mention that the computation of the delta itself takes time too.

Being aware of the limitations of delta-encoding, Douglis et al. designed a macro-encoding language called HTML Pre-Processing (HPP) for dynamic content delivery [DR97]. The authors observe that for each common class of resources such as Web pages provided by the same category in a Web server, a significant part of these pages remains static. Therefore, Web pages could be separated into static and dynamic parts. The static portion contains macro-instructions for inserting dynamic information. The static portion together with such instructions (called *template*[DR97]) could be cached freely. In other words, part of the dynamic information is cached as static. The macro-encoding language itself is an HTML extension. Some useful tags and keywords are added to HTML. For example, the tag LOOP is used to compress the repeated representation of content in a Web page. In this way, HPP not only allows the dynamic portions of similar pages to be transferred without sending the static portions, but allows a compact representation of repetition within a resource. Recall that for delta-encoding, server and the cache manager must agree on a common base version against which to apply a delta. HPP solves this problem by permitting a single cached template that all clients can cache, and providing additional dynamic information based on that template [DR97].

2.1.7 Negative Caching

Negative caching, as the name indicates, refers to prevention of caching activity by the origin server. In a Web-based system, when a Domain Naming System (DNS) look-up fails or a Web object retrieval failure occurs, it probably means that the remote server or one particular portion of the network is not available, and won't be available for at least a while. In this case, any immediately subsequent request to the same server might result in the same failure notice, therefore imposing unnecessary traffic on the network. Chankhunthod et al. propose the concept of negative caching [CDN$^+$96] where, when a Web object retrieval failure occurs, the caching manager caches the negative result for a parameterized period of time (e.g., five minutes), preventing any request to the origin server within this period. Any such request will result in the cache manager sending the client a failure notice right away.

2.2 Existing Web Caching Servers and Systems

The word *caching* has different meanings depending on the context. The purpose of using caching, whether it is in the area of computer architecture [PHG96], distributed file systems [SGK$^+$85], or distributed database systems [ÖV99], is mainly to make better use of available resources to speed up access. Over the past decade, significant research effort has been made in Web caching, resulting in a number of Web server application systems that contain caching functionality.

As mentioned in [CDN$^+$96], the **Harvest** software, which contains a hierarchical caching architecture, was developed with the aim of making efficient use of the network and of individual servers. Harvest project was the first to introduce the ICP protocol for communication between caches. **Squid** is a well functioning and full-featured Web proxy cache. It could support caching protocols such as ICP, CARP, and Cache Digests [RW98]. **CERN** Web server was developed by the World Wide Web Consortium (W3C) as a single server process that creates a new child process to handle each user request. This pretty much limits the scalability of the server itself. Other freely available Web server systems include Jigsaw[2], Apache[3], etc.

Web caching technology has also gained commercial importance. Companies like CacheFlow[4], CacheArray[5], and Fireclick[6] develop and sell Web caching technology

[2]http://www.w3.org/Jigsaw/

[3]http://www.apache.org

[4]http://www.cacheflow.com

[5]http://www.cachearray.com

[6]http://www.fireclick.com

in their integrated application software. Their customers include many e-commerce enterprises to whom reducing client latency means increased customer base and revenue.

2.3 Cache Consistency Algorithms

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 *C/S Interaction*. In client validation approach, it is the client (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

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

Table 2.1: A Classification of Cache Consistency Algorithms

weak cache consistency. We have briefly discussed these two terms in Chapter 1. As defined in [CL98], **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 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 2.1 gives a two-dimension classification of most publicly known cache consistency algorithms. Unfortunately, in the category of "C/S interaction-weak", we did not find any algorithm available to be put in. We discuss these algorithms in more detail in this section. We shall keep in mind that the relative performance of a consistency algorithm is very much dependent on the application system architecture and specific Web behavior. There is no single best algorithm that fits all scenarios.

2.3.1 TTL (Time-To-Live)

Under TTL approach, each object (document, image file, etc.) is assigned by its origin server a time-to-live (TTL) value, which could be any value that is reasonable to the object itself, or for the sake of 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 [RS02], 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 trade-off 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. The chance that this would happen is higher for a larger TTL. In [GS96], the authors initially use a flat lifetime assumption for their simulation, which means that they assigned all objects equal TTL values. This is also called *explicit TTL*, which results in poor performance. This was later modified and the TTL value was based on the popularity of the file. This method is also mentioned in [CL98], 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 [Cat92]. 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 on reasonable bounds ($<5\%$) [GS96].

TTL algorithm and its variations all enforce weak cache consistency, since it is possible that the cache manager satisfies client requests with stale object. 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.

2.3.2 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 the 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 [Cat92] 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 update threshold times the object's age [GS96]. For example, consider a cached file whose age is 30 minutes and whose validity was checked one minute ago. If the update threshold is set to 10%, then the object should be marked as invalid after 3 minutes (10% * 30 minutes). Since the object was checked the minute before, requests that occur during the next two minutes will be satisfied locally, and there will be no communication with the server including control message. After the two minutes 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.

2.3.3 Invalidation

The Invalidation algorithm frees client cache manager from the burden of sending If-Modified-Since (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 don't need to worry about object validity. As long as 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 burder for the Web server, because, in order to invalidate, for each single object that has ever been accessed, the server has to keep 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 or never requested again, or the cache manager needs free space for newly-arrived objects. Therefore, it does not make sense for the server to keep 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 updates* [RS02]. However, this approach might be a big problem if one of the caches loses contact with the network and becomes unreachable. The server will have to keep waiting and none of the caches 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 [YBS99]. 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 [RS02], 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 life becomes miserable if it sends out an invalidation message to such an unavailable client and keeps waiting for acknowledgement from that client before updating Web pages.

2.3.4 Piggybacking Approaches

So far, the algorithms we discussed are all synchronous, i.e., after the initiating part (either client or server) sends out a validation/invalidation message, it will wait till it has received a response/acknowledgement. This sometimes introduces extra and unnecessary time delay. The reason is that for client validation, in many cases, the object on the origin server has not been changed and the validation request will simply result in a "Not Modified" response. For server invalidation, the message that the server sends out might not be critical to certain caches. In other words, if the cached object will not be requested for quite a while, it is unnecessary for the server to send out every single invalidation message once an object update is going to happen. If the server-update is often, these invalidation messages will be overwhelming, which is inevitable if strong consistency is the purpose. On the other hand, if strong cache consistency is not a requirement, we might use asynchronous mechanisms to do the job. Krishnamurthy and Wills [KW97, KW98] introduce algorithms using piggybacking to improve cache consistency while reducing network traffic resulted from validation/invalidation control messages. Piggyback Client Validation (PCV) is another client-driven algorithm, and Piggyback Server Invalidation (PSI) falls into server invalidation category. Obviously these two algorithms are both weak due to their asynchronous nature.

Piggyback Client Validation (PCV) [KW97] is an enhancement of traditional TTL mechanism. Instead of sending IMS messages to the server every time the cache manager receives a request for an object whose TTL has expired, the validation is done in a different way. Whenever a certain message is going to be sent to a server, the cache manager piggybacks 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.

Piggyback Server Invalidation (PSI) [KW98] 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.

On the other hand, Piggybacking mechanisms introduce some side-effects as well. For PCV, the cache manager has to keep a piggyback list for each server in order to conduct the piggybacking operation. This requires extra memory/disk space, which may slow down the validation process. Such extra delay also exists in PSI approach. As we mentioned, all piggybacking approaches could only enforce consitency to a certain level. Stale hits will still happen. Krishnamurthy and Wills propose a hybrid scheme, which combines PSI with PCV [KW98]. They claim that the hybrid scheme can result in stale hit ratio of 0.1%. However, we did not put this hybrid algorithm into the "C/S interaction-Weak" category of our classification table. Designing a consistency algorithm that falls into this category will be a very interesting research topic, and piggybacking could be a starting point.

2.3.5 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 [GC89]. 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 indefinite 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 doesn't mean that the longer a lease is, the 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 [YADL98, YADL99]. 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 vulume it belongs to. The cache manager cannot respond to 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.

2.4 Prefetching

The idea of *prefetching* Web pages comes as the result of long response time that user experiences. Usually it takes seconds or more to download a page in a normal-speed Internet connection environment. It would be nice if documents or Web pages that are likely to be requested by a client could be retrieved in advance. Examples are images that are in a document, and other documents pointed to by hypertext links from within a requested document. Despite expected performance improvement, retrieving all documents referred to by a given page is not a wise choice either [Luo98]. It would cause bursts of requests, and if this were done for every incoming request, it would reverse the effect of proxies from being bandwidth savers to bandwidth burden consumers retrieving everything possible that a user might request.

Prefetching is sometimes used in the context of consistency control, too. When a cache manager realizes that some of its cached objects are stale because of their TTL expiration, it might contact the origin server to validate these objects or retrieve the updated version at an off-peak time, e.g., during the night or network idle period. This potentially improves cache consistency while making better balance of network bandwidth.

2.5 Cache Replacement Algorithms

If a cache had infinite storage space, the cache manager would simply store all the requested objects in cache and never remove any of them unless the copy on origin server has been changed. This would be nice, because it maximizes the possible cache hit rate. However, the reality is that the storage space (main memory or disk) is limited, and if the cache is active, sooner or later it will get full. As a result, any practical Web caching system has to deploy certain replacement algorithm to make best decision on which object(s) to evict when it is inevitable to do so.

There are several factors that affect the performance of cache replacement algorithms: object size, object retrieval time, access popularity and frequency. If every object is considered equal in terms of these factors, a simple queueing model, or First-In-First-Out (FIFO) rule would be more than enough for replacement policy. Again, the reality is more complicated, in that different Web objects vary significantly in the above-mentioned factors. Using FIFO as replacement algorithm is feasible, but might result in poor performance because it does not consider the feature of Web object differences.

The most classic replacement algorithms include Least Recently Used (LRU) and Least Frequently Used (LFU). As their names indicate, LRU replaces the object that was used least recently and LFU replaces the object that was used least frequently. Both algorithms can be implemented easily and are actually used in many areas such as computer architecture and distributed file systems. In the context of Web caching, LRU is usually used as the basis of comparison when new replacement algorithms are proposed.

Despite the advantage of simplicity that LRU has, numerous experiments prove that the performance of basic LRU in Web caching is poor [ASA$^+$95, CI97, JB00, PP01]. The reason is that LRU only considers the access frequency of Web objects while neglecting two other important factors, object size and retrieval time delay. If we are only concerned about object size, we can use a largest-size-first replacement policy, which always removes from cache the object with largest size. For the workloads studied by Williams et al. [WAS$^+$96], the largest-size-first algorithm achieved the highest cache hit rate, but performs worst in terms of byte hit rate.

Similarly, some cache replacement algorithms only focus on retrieval latency. Lowest Latency First algorithm [WA97] first replaces the objects that can be retrieved faster from their origin servers. The object that takes more time to download are less likely to be replaced. This set of algorithms, like the above-mentioned algorithms that focus only on one factor of the many, might perform well in certain workload, but won't fit other scenarios. Researchers realized that any of the factors that would affect caching performance should be taken into account while designing cache replacement algorithms. Abrams et al. introduced LRU-MIN and LRU-Threshold [ASA$^+$95] that are based on the strict LRU but consider the other factors. LRU-MIN selects from a list of objects that are sorted by LRU and remove the large object first. This reduces the number of objects that need to be removed. LRU-Threshold only caches object that are smaller than a certain size limit. In this case, large objects such as multimedia documents whose sizes are normally in the order of tens or hundreds of million bytes would never be cached. In a workload where large-size objects are accessed frequently, this algorithm won't be winning either.

The Lowest Relative Value (LRV) algorithm [RV00] includes the retrieval time and size of an object in calculating its priority of being kept in the cache. The algorithm purges the object with the lowest value. The Greedy-Dual-Size algorithm proposed by Cao and Irani [CI97] is similar to LRV but more flexible in that it calculates such value depending on whether the goal is to maximize hit ratio, to minimize average latency, or to minimize the overall retrieval cost. The Greedy-Dual*Web Caching algorithm [JB00] takes this approach one step further by considering the fact that objects that belong to the same Web page and were retrieved together will most likely be retrieved together in the future. Therefore, if there is a cache hit on any of the objects in a Web page, all of them should be kept in the cache.

Exploring all the cache replacement algorithms is outside the scope of this thesis. On the other hand, no commercial vendors of caching system distinguish their product by the replacement algorithm they use, simply because cache replacement policy is not as critical a factor in the performance of Web caching systems as cache consistency algorithms. As a matter of fact, the storage space is no longer as major a concern as it used to be.

2.6 Performance Measurement Tools

As pointed out by Rabinovich and Spatscheck [RS02], performance evaluation methods could be generalized into three main categories: live measurement, trace-based methods, and benchmarking. Live measurement is based on the real-time Web activity. Experiments are conducted in an active Web environment. The advantage of this is that it reflects one set of real Web transactions that actually happened. The drawback of using a live measurement, though, is that there is no way to implement such a real-time Web behavior, not to mention that such live measurement is not repeatable. It is not suitable for stress test either, because nobody wants to overload their system in the real case, in other words, some situations that researchers want to explore will never happen in a live measurement.

As a matter of fact, a lot of research in Web caching literature is based on trace-based methods, i.e., getting the access log of a certain period of time from the server or proxy, and replaying it to analyze the Web behavior. It is not hard to configure the Web server or proxy to get a trace log that is designed for a specific research, e.g., time and sequence of each URL request, size of each object, response status code, etc.. This makes the research work easier, partly because the experiments could be repeated as needed. On the other hand, Web behaviors differ significantly in terms of different Web servers and end user groups. Therefore, the experiment results obtained from a certain trace log might only fit one specific access pattern or commercial scenario.

Another category of measurement methods is by using benchmark. Benchmarks are synthetic workloads designed to mimic real-life workloads [RS02]. Specific benchmarks are used for specific academic or commercial purposes. In the context of Web caching, the performance of caching proxies is the most widely studied topic and quite a few benchmarks have been proposed as the standard for comparison. The Wisconsin Proxy Benchmark was developed in an attempt to provide a tool to analyze and predict performance of different proxy products in real-life situations [AC98]. Another product is Web Polygraph, a freely available benchmarking tool for Web caching proxies. Polygraph distribution includes high-performance Web client and server simulators. Polygraph has been used to test and tune most leading caching proxies.

Since the focus of our research is the performance comparison of different cache consistency algorithms, we use TPC-W benchmark because we want to explore the cost of keeping strong consistency in an e-commerce environment where object updates are often and object freshness is mission critical. TPC-W is a transactional Web e-commerce benchmark and a member of industry and academy accepted benchmarks designed and supported by he Transaction Processing Performance Council (TPC). TPC is a non-profit organization founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry[7]. We will discuss TPC-W benchmark in a great more detail in the next chapter.

[7]http://www.tpc.org/information/about/about.asp

Chapter 3

TPC-W Benchmark

This chapter gives a brief introduction of TPC-W benchmark. Interested reader should refer to TPC Web site[1] for detailed specification and/or industry results.

3.1 Benchmark Description

TPC Benchmark W (TPC-W) is a transactional Web benchmark [Tra01] that simulates the activities of a 24/7 online bookstore. The benchmark measures both business-to-consumer (B2C) and business-to-business (B2B) models. It includes real world features such as security, shopping carts, credit card validation, load balancing and Web page information from the database. The original purpose of developing this benchmark is to have a standard e-commerce workload to measure and compare the performance of specific components of a commercial online business system, such as Web server, database server, or application server. It is

[1]http://www.tpc.org

desired by customers, who are considering buying hardware/software package for their e-commerce solutions, that most vendors of such systems test their products on TPC-W benchmark and publicly announce their performance results.

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 configured is defined as the scale-factor of the benchmark. We will discuss in more detail the scaling requirements of the benchmark 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, 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 are represented as Web pages on the bookstore Web site. Figure 3.1 gives an example of how an EB might move through the

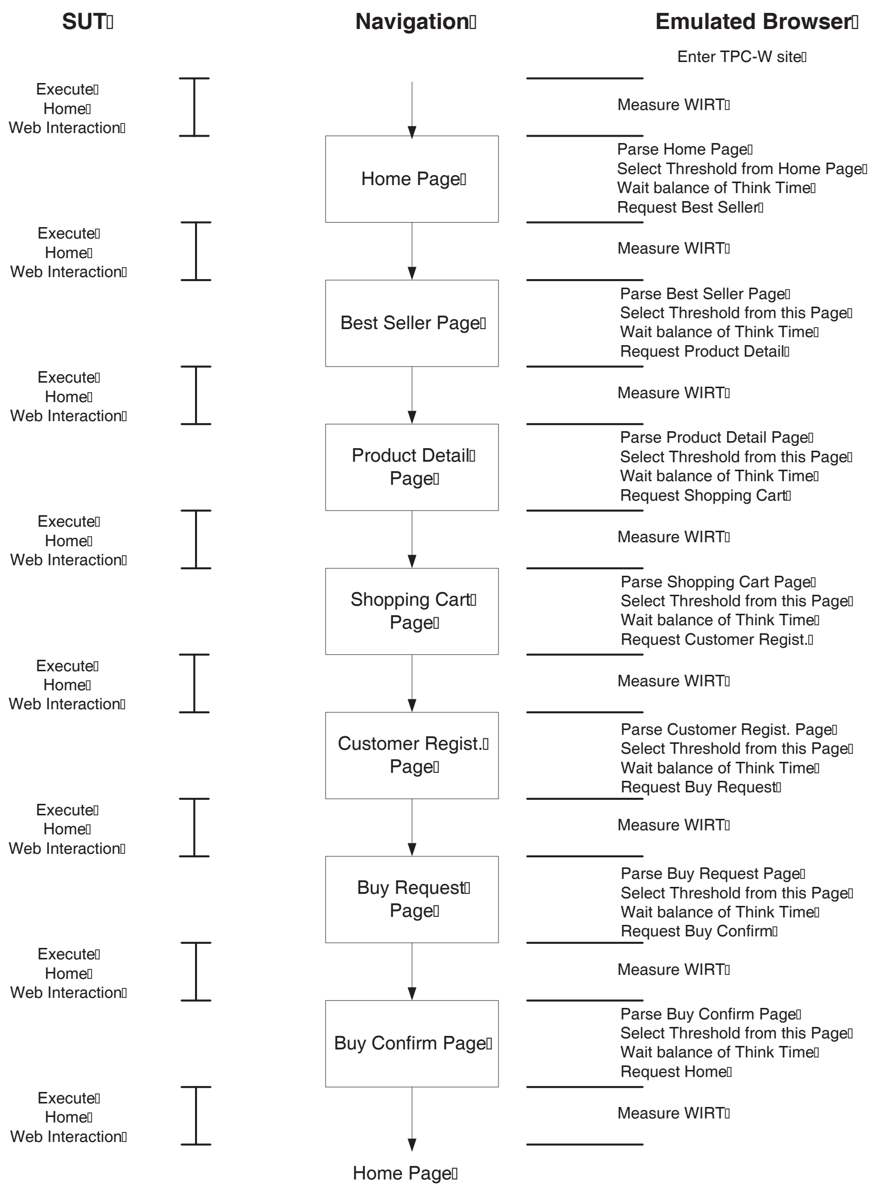

Figure 3.1: An Example of EB activities at TPC-W Web site (adapted from [Tra01])

TPC-W Web site, and the interactions between the System Under Test (SUT) and EB. The usage of TPC-W benchmark is not limited to emulated browsers, which is part of a computer program. We have our implemented benchmark reside at URL: *http://gocek.uwaterloo.ca/tpcw/servlets/TPCW_home_interaction*. This implementation was originally done by a course project team at University of Wisconsin-Madison. The detailed discussion of the benchmark implementation is presented in Chapter 5.

Obviously, the fourteen Web interactions impose different workloads on 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 System Under Test (SUT), which is the collection of servers that support TPC-W e-commerce interaction.

The users who come to visit the bookstore Web site might have different interests. Some might be just browsing, searching for interested 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 could be found in the next chapter.

3.2 System Architecture

Before looking into the scaling requirements of TPC-W benchmark, it is helpful to understand the functionality of components that constitute a valid TPC-W environment.

3.2.1 Remote Browser Emulator

The Remote Browser Emulator (RBE) is the software component that drives the TPC-W workloads [Tra01]. The purpose of the RBE is to drive the System Under Test (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 [Tra01] 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. Readers of this thesis can refer to the benchmark specification for the calculation formula of USMD [Tra01]. Like a normal Web user, the EB can start a new user session right after it terminates one. This is controlled by RBE, too.

3.2.2 System Under Test

The System Under Test (SUT) is the integrated part on which we conduct performance evaluation. As displayed in Figure 3.2, 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 TPC/IP connections, is also regarded as part of SUT.

Although performance test could focus on certain component of SUT and only care about that part, each component of SUT as shown in Figure 3.2 must exist to perform a valid run. The SUT performs all the operations required for successful Web interactions, such as required database access and communication between EB and Payment Gateway Emulator (PGE).

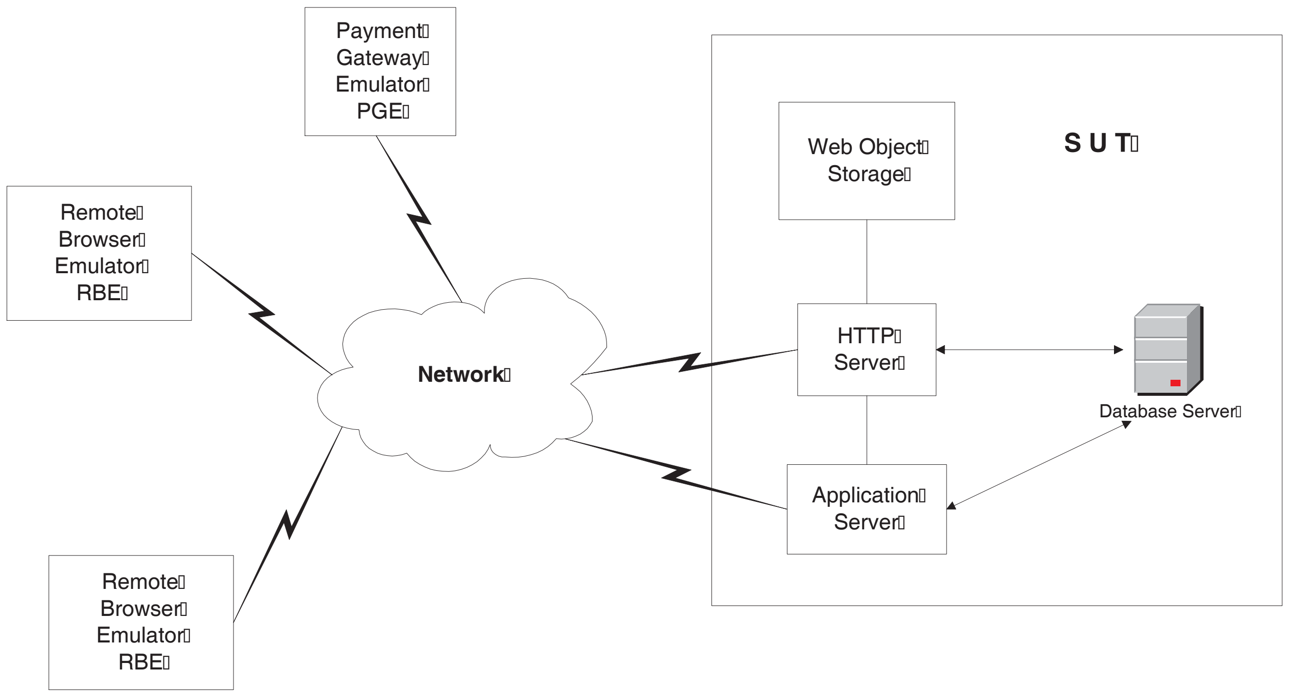
Payment Gateway Emulator PGE
Remote Browser Emulator RBE
Network
Remote Browser Emulator RBE
S U T
Web Object Storage
HTTP Server
Application Server
Database Server

Figure 3.2: Model of the Complete Tested System (adapted from [Tra01])

3.2.3 Payment Gateway Emulator

The Payment Gateway Emulator (PGE) represents an external system that authorizes payment of funds as part of purchasing transactions [Tra01]. 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. In a real e-commerce Web site, this component is critical and indispensable for customer to conduct secure payment transactions. However, whether the PGE performs well or not is trivial to our experiments. As long as we simulate the response time delay properly, we can get around the absence of a PGE. In our experiments, we don't have a full-functioning PGE. Instead we always assume that the information customer entered is valid and will receive positive acknowledgement from PGE so that the EB can go ahead and complete the ordering transaction. This is a simplifying assumption that we made to keep us from extra effort of developing such an emulator and at the same time, our experiments are still valid because PGE is not part of SUT. Meanwhile, we have reasons to believe that PGE component is not mandatory to the benchmark. More discussion will follow.

3.3 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 [Tra01]. 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 too. Obviously, the configured EBs must remain active and generate Web interactions throughout the entire measurement interval [Tra01].

Besides the number of EBs, database size is another scaling factor that will affect throughput. According to the benchmark specification [Tra01], 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; in other words, the performance result under a database size of 1,000 items cannot be compared with that of a database containing 10,000 items.

Let's take a look at the database component of SUT. There are altogether 8 tables that are required by TPC-W: **Customer, Country, Address, Orders, Order_Line, Author, CC_Xacts** and **Item**. The detailed field description of these tables could be found in Appendix B. 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 inequalities:

$$\text{(number of EBs)}/14 < \text{WIPS} < \text{(number of EBs)}/7$$

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 close to the lower bound conceptually. 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 tester to verify whether the SUT has been over-scaled.

3.4 Performance Metrics

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. texts, forms, images, etc.), which, when put 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.

Besides WIPS, the benchmark also measures the response time of each successful Web interaction, i.e., 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 [Tra01]. As we discussed in Chapter 2, response time, or client latency, is an important performance indicator of a Web caching system. It shows whether the extra delay due to the introduction of a cache consistency or replacement algorithm is kept to a minimum. It also shows whether the caching architecture would have adverse impact on the overall system performance. Definitely, WIPS can also measure that, because it indicates the system throughput.

Another term that is defined in the benchmark specification is user think time. It is the period between the time when the last byte of the last Web interaction is received by the EB from the SUT, to the time when the first byte of the first HTTP request of the next Web interaction is sent by the EB to the SUT [Tra01]. Like USMD, the value of each think time must be taken independently from a negative exponential distribution. Refer to the benchmark specification for detailed computation method [Tra01]. The average duration of the think time over each measurement interval, for each type of Web interaction, must be no less than seven seconds and no more than eight seconds.

The performance of the SUT under a constant overload state is also measured by TPC-W. This test helps demonstrate system behavior when it is driven with more than the normal load. For a Web server or an online application server, this is likely to happen when many users visit the Web site at the same time, or certain portions of the Web site suddenly becomes popular.

3.5 Benchmark Implementation

Since the first version of TPC-W benchmark specification was publicly announced in 1999, the benchmark has been implemented using different programming languages on various platforms. We considered two implementations: one by IBM Toronto Research Lab on Linux system, which was written in C++ language, and a pure Java version implemented by a team at University of Wisconsin-Madison.

The former implementation was not publicly available, while the latter was online with complete source code and installation guide. Considering the compatibility advantage of Java implementation, we chose the latter package, and implemented all our cache consistency algorithms on top of it.

3.6 Why TPC-W

The reason we chose TPC-W to study cache consistency algorithms is because TPC-W is a Web commerce benchmark and it generates workloads that can involve frequent object update at server side. It models a real-world e-commerce Web site, whose activity and performance is of great interest.

Chapter 4

Evaluated Consistency Algorithms

Cao and Liu [CL98] compare the performance of three cache consistency algorithms: adaptive TTL, polling-every-time and invalidation. The first one enforces weak cache consistency, while the latter two maintain strong consistency. Polling-every-time, as a strong consistency algorithm, is known for its inefficiency, because, under this algorithm, the number of IMS messages reaches the maximum. In our research, we are more interested in strong cache consistency mechanisms. Besides Polling-every-time and Invalidation that were examined in [CL98], we add Lease algorithm as per our classification in Table 2.1. We also evaluate and compare the performance effect of TTL algorithm, since it is the most popular consistency algorithm in the "weak" category. Our evaluation of these algorithms is done within the framework of a recognized benchmark.

We are interested not only in consistency 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 our research. We implement each of the examined algorithms both in a browser cache and a proxy cache, and compare the result horizontally (per algorithm) and vertically (per location). We implement all these algorithms on top of the Java implementation of TPC-W benchmark from University of Wisconsin-Madinson[1]. In order not to change the benchmark code, we follow the class definition that was already in the existing Java code. We add two classes, CacheObj and Cache, for each caching algorithm, and all the necessary operations according to individual algorithms. They will be discussed in more detail in following sections.

4.1 Cache Replacement Algorithm

Any real-world caching system needs some replacement mechanism because of its limited storage space. In order to implement a functioning cache for our research, we need to choose a replacement algorithm as well. Since the focus of our research is not replacement algorithms, we just use the widely accepted LRU-SIZE algorithm [RS02] as our replacement policy. The algorithm sorts cached objects according to their access frequency and size. The least recently used objects will be evicted first, and if there are two or more objects that are equally least recently used, the one with the biggest size will be evicted first.

In our implementation (see Figure 4.1), we separate each cache into two storage

[1]www.ece.wisc.edu/~pharm/tpcw.shtml

```
CachedURLs    :    an array that stores the information of all cached Web pages
NewSize       :    the size of the new object to be stored in cache
removeSize    :    the size of the object to be removed from cache
CACHELIMIT    :    the storage limit of the cache
CurrentTotal  :    current total storage size of all the cached objects
K             :    the index of the object to be removed from cache
maxAge        :    the access frequency of an object, the less frequent, the bigger


if (CurrentTotal + NewSize) < CACHELIMIT
        return;            //The cache still has enough space, no need for replacement

while(CurrentTotal + NewSize > CACHELIMIT)  {
        let i = 1;  // point the cursor to the first element of CachedURLs
        let K = 1;
        let maxAge = 0; //initialize
        while it's not the end of the cache list  {
                if the age of object CachedURLs(i) is bigger than maxAge  {
                        assign the age of CachedURLs(i) to maxAge;
                        let K = i;
                        assign the size of CachedURLs(i) to removeSize;
                }
                else if the age of object CachedURLs(i) is equal to maxAge  {
                        if the size of object CachedURLs(i) is bigger than removeSize  {
                                assign the age of CachedURLs to maxAge;
                                let K = i;
                                assign the size of CachedURLs(i) to removeSize;
                        }
                }
        }

        remove object CachedURLs(K) from cache;
        CurrentTotal = CurrentTotal - removeSize;
}
```

Figure 4.1: Pseudo-code of LRU-SIZE cache replacement algorithm

sections, one for HTML pages and the other for images. There are two reasons for this arrangement. First, we want to speed up cache object search. By grouping, the cache manager doesn't need to search Web pages in image objects, nor does it need to search images in a long list of Web pages. Second, since the size of image files is normally much larger than HTML pages, we don't want the replacement algorithm to always remove image files from the cache. The replacement takes place only when the total size of the two sections reaches cache capacity. In other words, these two storage sections share the total cache. If a new image file needs to be stored in cache but there is not enough space, then the cache manager will first remove the objects in the image storage section, and if there is still not enough free space after all images have been evicted, object removal will take place in the HTML page section. However this situation will never happen unless the cache size is so small that it fills up with objects that are all HTML pages (because images are always retrieved after HTML pages).

4.2 Infinite Caching

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 give us the upper bound of cache hit ratio and WIPS throughput.

We are also aware that infinite caching will not always give upper bound performance result although conceptually it should. 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 time for the cache manager to search through the cache to make decision on whether it is a cache hit or cache miss. 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. In our research, we are not going to find such a threshold, because even if the threshold does exist, it might take extremely long time (to fill up a big-enough cache storage) to get to that state.

4.3 Time-To-Live

We make the simplifying assumption that image files will never expire, therefore we don't worry about the TTL value of images. For the TTL values of HTML Web pages, we follow what is required by the benchmark specification, i.e., we set the TTL value of most pages to be 30 seconds. For those pages that don't require validity check every 30 seconds, we set their time-to-live to be 30 seconds to enforce stronger consistency. Similar to polling-every-time, we created a server-side Java class called TTLSocketManager that keeps track of updated pages. This socket manager handles both page-update messages from serlvets and IMS messages from EBs.

Figure 4.2 gives the pseudo-code of TTL algorithm implementation.

4.4 Polling-Every-Time

By now, it's conceptually clear that polling-every-time (POL), although maintaining strong cache consistency, is inefficient in the sense that a major portion of validation messages might turn out to be just unnecessary. However, it could be beneficial and efficient if used in an environment where most objects are of large size. This will give us a great saving by sending IMS messages instead of complete document transfer. In TPC-W benchmark environment, the size of most HTML documents are around 5-10KB. Image files, which usually have size of 250KB, don't need to be polled because their validity is always assumed. On the other hand, the

```
Upon receiving the client URL request, check the cache to see if the page is in cache;

point the counter to the first URL in cache;
for all the cached pages, do
  if the URL of current page is the same as the requested URL  {
          if the TTL of the current cache page has not expired  {
                    return the corresponding Web page to the EB;
                    set the access frequency value of current object to 0;
          }
          else  {
                    send If-Modified-Since message to Server ;
                    wait till reply from server is received;
                    if the URL has expired  {
                              inform EB to retrieve the Web page directly from origin server
                    }
                    else  {
                              set the access frequency value of current object to 0;
                              return the page to EB;
                    }
                    break out of the for-loop;
          }
  }
  else  {
          increase the access frequency of current object by 1;
          go to the next cached page;
  }

endfor
```

Figure 4.2: Pseudo-code of Time-To-Live algorithm

size of an IMS message is in the order of bytes, therefore, we still expect that polling-every-time performs better than no cache.

From the perspective of implementation, it is not hard to send IMS message to server every time an EB submits a URL request. At server side, we need to record any database changes as the result of Web transactions. Therefore, we created a server-side Java class called PollSocketManager, which keeps a list of updated pages and their update time. We studied TPC-W benchmark carefully and found that only the shopping cart operation could result in database update. Therefore, we changed the shopping cart servlet to have it notify the socket manager of any page updates so that the socket manager updates its list accordingly.

We implemented POL by making slight changes to TTL algorithm. Instead of checking whether the TTL value of the cached object has expired, whenever a cache hit happens, the cache manager sends out an IMS message immediately to validate the freshness. From implementation perspective, POL has the advantage of simplicity.

4.5 Invalidation

As discussed in Chapter 2, invalidation algorithm requires server side bookkeeping. The most strict invalidation approach requires that the server send out invalidation messages, wait for acknowledgement notice from all the parties to which the messages were sent, and then proceed with the update operation. As we argued, the delayed update does not make much sense because the semantics of the Web page has changed anyway.

For invalidation algorithm, we changed the shopping cart servlet to have it send out an invalidation message as soon as it is triggered, and the update operation is not delayed. Notice that the servlet sends out only one message at a time, and this message is sent to a version control manager that resides on RBE. Upon receiving this message, RBE will go through all the EBs that it manages and remove all the objects that each EB has which become out-of-date due to the update operation that has taken place on the server. Again, at this point proxy cache has an advantage over client cache, because in the case of proxy caching, the version control manager needs to evict objects from only one cache, compared to browser caching.

The content of the invalidation message is very simple, just the item id. Upon receiving the invalidation message, all the Web pages that contain this item id will be deleted. In the real world, however, this might be unnecessary, because even some portion of the item information is changed in database server, the Web pages that contain general information of the item would still be up-to-date and not affected by such update. Here we make a stronger assumption that once any part of an item information (e.g., title, amount available, unit price, etc.) gets changed, all the pages that contain any information about this page will be out-of-date, and, therefore, cached copies should be evicted.

The version control manager extends the role of server as an invalidator. It is similar to the socket manager classes we implemented in polling-every-time and TTL, except that it resides at proxy.

4.6 Lease

As we discussed in Chapter 2, it is a bit tricky to assign appropriate lease values to Web objects. If it is too long, server has to wait longer before an update can take place. But if it is too short, more IMS messages will have to be sent. In our implementation, we treat all the online store book items equally and assign the lease value to be 20 seconds, a bit shorter than the standard time-to-live value in TTL algorithm. This might result in a few more IMS messages than TTL, but it is expected to perform better than POL while keeping strong consistency.

We created a Java class called LeaseServerManager, which resides on the server and keeps track of all the requested item IDs. When the server is about to make changes to a certain item, it checks if that id is in the request list, if so the current server process will idle until the lease for that id has expired. The process on client side is similar to that of TTL. As long as the lease on an object is still valid, the cache manager could satisfy user requests by simply return the cached copy without validating the object. Once it gets expired, the next hit will result in the validity check.

Figure 4.3 gives the pseudo-code of Lease algorithm implementation.

```
Server Side:
 if a servlet is going to issue update statement to the database  {
      send a message with the item ID to LeaseServerManager;
      if there is a valid lease for that ID  {
         keep current servlet sleeping until the lease expires;
      }
      issue the SQL statement to database;
  }


Client Side:
 upon receiving the client URL request, check the cache to see if the page is in cache;
 for all the cached pages, do
    if the URL of current page is the same as the requested URL  {
         if the lease of the current cache page has not expired  {
                  return the corresponding Web page to the EB;
                  set the access frequency value of current object to 0;
         }
         else  {
                  send If-Modified-Since message to Server ;
                  wait till reply from server is received;
                  if the URL has expired  {
                           inform EB to retrieve the Web page directly from origin server
                  }
                  else  {
                           set the access frequency value of current object to 0;
                           return the page to EB;
                  }
                  break out of the for-loop;
         }
    }
    else  {
         increase the access frequency of current object by 1;
         go to the next cached page;
    }

 endfor
```

Figure 4.3: Pseudo-code of Lease algorithm

Chapter 5

System Model and Experiment Setup

5.1 System Model

In this section we introduce the system model that we set up for the performance study. 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 5.1 is an overall picture of our system model.

5.1.1 Client

According to the benchmark specification, the RBE acts as the driving program for the benchmark workload. It creates multiple threads during runtime, which

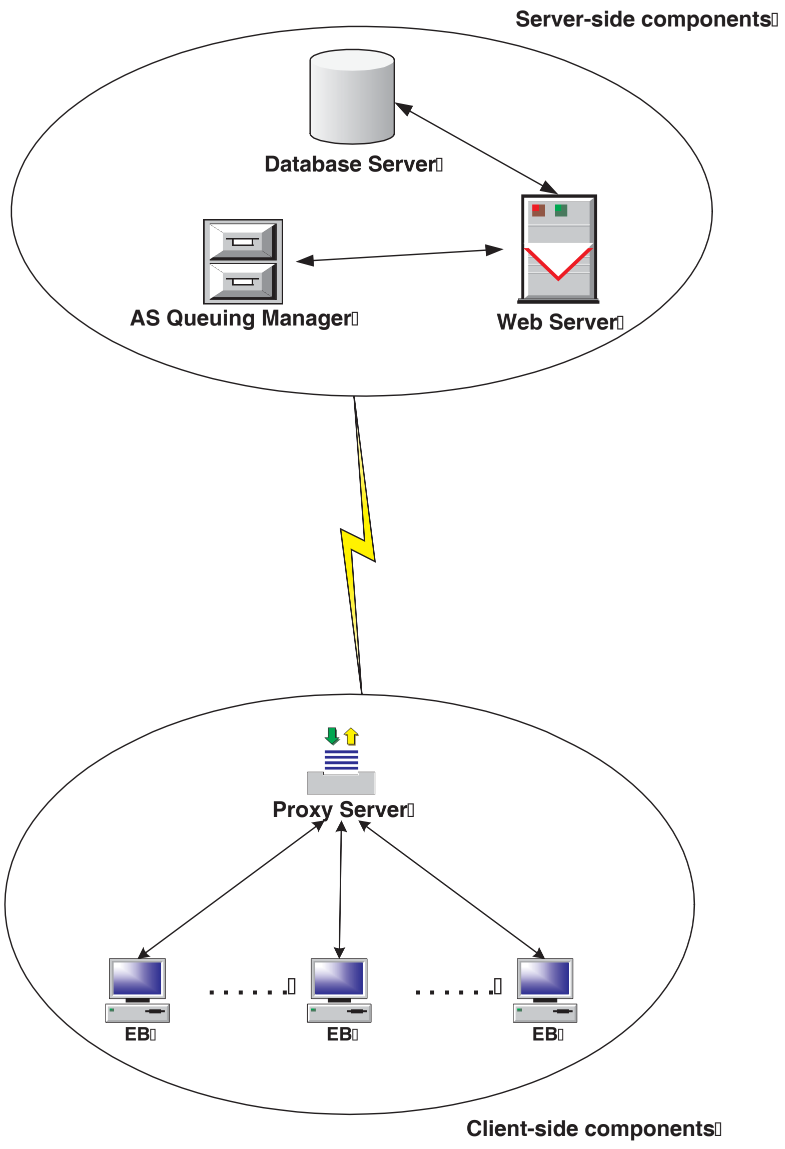

Figure 5.1: Simulation System Model

simulate real Web users who browse the online bookstore Web site and place orders to buy books. The user thinktime, as introduced in Chapter 3, is calculated by the RBE. Individual EBs are not allowed to communicate or share information directly with each other.

5.1.2 Proxy Server

Proxy server is not mandatory for an E-commerce Web site, because Web interactions don't need to go through proxies. However, we have the proxy server component in our system 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. We would rather have a proxy server component here to give RBE a physical location in the architecture. We shall point out that we did not implement a full-fledged proxy server. Instead we implemented 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 will force all currently-running emulated browsers to stop and produce statistics reports based on information gathered during the measurement interval.

5.1.3 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. According to the benchmark specification, the database contains 8 master tables (see Chapter 3). To build up the whole set of data, we use a database population program that is shipped with the Wisconsin Java implementation kit. When we populated the database, we assume that there are at most 100 concurrent users accessing the online bookstore, and the bookstore sells 100,000 book items. These are the parameters required by the database population program.

5.1.4 Web Server

Since our main objective is to compare performance of cache consistency algorithms in a Web e-commerce environment, our criteria of choosing the Web server is one that is widely used, easy to configure and easy to use. The Wisconsin kit is claimed to have been tested under various Web Servers such as Java Web Server, Jakarta Tomcat, Jigsaw Web Server, etc. We set up the environment with Jakarta Tomcat 3.2.1, and found Tomcat to be pretty easy to use and robust. Although it is not a fully functioning HTTP server, it's powerful enough for our experiments, 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 on 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. For example, if 4 EBs are requesting the same Web page at the same time (or within a very short time period), there will be 4 concurrent connections to the same servlet on the server. The capability of servlet container on the Web server determines how many concurrent requests it can handle without crashing the server. Commercial Web servers should generally have much more powerful servlet containers than Web servers that are free of charge, such as Jakarta Tomcat and Jigsaw. Apache, another freely available Web server, is more powerful than Tomcat, but it still has to use Jserv or Tomcat as its servlet container component.

We also tried to use Java Web Server and Jigsaw. We visited Sun Microsystems Inc.'s Web site, and found out that on February 7th 2001, Sun announced the end of life of Java Web Server[1]. Using this product might result in lack of support of new technology and protocols in the future. Jigsaw Web server is W3C's Web server platform. However we found that it is not as easy-to-use as Jakarta Tomcat, although it takes much more effort to install and configure Tomcat rather than Jigsaw. For the above reasons, we still decided to use Jakarta Tomcat 3.2.1 as our Web server and stand-alone servlet container. This results in its performance limitation in scalability that we have to get around within our implementation. On the other hand, stand-alone servlet containers perform much faster than out-of-process servlet containers, which have better performance in many other measurable ways such as scalability and stability.

5.1.5 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

[1]http://www.sun.com/software/jwebserver/index.html

that prevent private customer information being stolen 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 kit from University of Wisconsin-Madison 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 paper written by Edwards et. al [EBH$^+$01]. The authors introduce a methodology that uses an analytic model to simulate e-commerce applications. This model is implemented using Lotus 1-2-3 spreadsheet. It considers page size, number of images in each page, whether or not using SSL, and whether applying cache as four factors that determine server response time. The basis of the model creation is a case study of ShopIBM Web site. We worked with one of the authors to determine the server response time and application server capability in our experiment environment. We adjusted the spreadsheet according to our scenario and found that the average response time of the application server should be 0.5 second. 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 seconds. 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 will be stored in a request queue first. In other words, if the server receives 50 requests at a time, 41 of them will be served immediately and the remaining 9 are placed in queue. The queuing manager sends acknowledgement back to client based on its service capability. We also add time delay during the servlets execution to simulate the application server response factor.

5.1.6 Network Connection

We made a simplifying assumption that the client side (proxy and individual Web browsers) and remote server are connected through high-speed network, i.e., we didn't simulate the data transfer exactly the same as real world. Our simulation model is that for each data transfer, the network delay time is decided by a randomly generated number between 1 to 3 seconds.

5.2 Experiment Setup

5.2.1 System Parameters

Hardware configuration

In our experiments, the server is a PC with Pentium III 1GHz CPU, 512MB RAM and 80GB 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, 128MB RAM and 30GB 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 100MB data transfer speed. Both of them run on Windows 2000 operating system.

Database Size (specified by TPC-W)

As discussed in Chapter 3, 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. The size of ITEM table, according to the benchmark specification, must be chosen from the set of defined scale factors as follow:

1,000; 10,000; 100,000; 1,000,000; 10,000,000.

In order to conduct our experiments under a database environment with representative size, we initially decided to have 100,000 items for the database, i.e., we want to simulate a mid-sized online bookstore. However, experiment results show that the time delay at server side was so long that the system throughput is far below our expectation. The reason for this is the limitation of Windows 2000 operating system itself. 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 250KB, and a thumbnail file of about 6KB. 100,000 items require a file structure of 100 directories, each directory containing 2,000 files (each item needs two image files), and the total size of all the images is approximately 26GB. We found that the time delay at searching image files in the file structure is significantly long. Several times the process went into "not-responding" state. Because of the hardware limitation, we changed the database size to have only 10,000 items. We also fixed the number of EBs to be 100 when populating the database. During experiments, we plan to still fix the item number (because it determines the size of all database tables, and the directory structures for storing images) as one of the runtime parameters of the benchmark, while varying the number of EBs in an attempt to see the impact of client population on system performance.

The number of EBs (specified by TPC-W)

Theoretically speaking, there is no limit on the number of EBs that we can specify to run the benchmark with. The more EBs we specify, 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 we apply client-side caching algorithms, since all our 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.

Web Interaction Mixes (specified by TPC-W)

The benchmark defines three distinct Web interaction mixes: browsing, shopping and ordering. Browsing mix involves 95% browsing and 5% ordering, shopping mix has 80% browsing and 20% ordering, while ordering mix involves 50% browsing and 50% 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.

Cache size

Except for the infinite cache, we specify the maximum cache capacity in the program. This means 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 100KB, and increase in increments of 100KB 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 reach the memory limit). This maximum value can be increased by increasing the available RAM on the testing machine.

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.

5.2.2 Performance Evaluation Metrics

Following are the performance metrics we use in our research to evaluate consistency algorithms.

Web Interactions Per Second (WIPS) (specified by TPC-W)

The number of Web Interactions Per Second (WIPS) is one of the two primary metrics of the TPC-W benchmark (the other one is a price performance metric defined as Dollar/WIPS, but it is not quite relevant to our research). Basically the value should be stable in a small range given fixed EB number, cache size and server response time, and it should increase with more EBs. But we expect a boundary value (maximum throughput) where the WIPS becomes stable even when the number of EBs increases dramatically.

Response Time (specified by TPC-W)

In the TPC-W benchmark specification, response time is measured per successful Web interaction and used as an indicator of how fast the System Under Test (SUT) could perform. In the Web caching literature, response time is one of the most important performance metrics for caching architectures and algorithms. During our experiments, we collect response time for every successful Web interaction. Such response time includes network delay and therefore corresponds to client latency.

Hit Ratio

In our experiments, we define *cache hit ratio* as the total number of bytes that all the EBs received from caches divided by 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. We keep a global count 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. We measure this result 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. It is possible for both directions.

Traffic

There are two types of measurement traffic in our experiments. The first is the total number and size of control messages. For TTL algorithm, there will be If-Modified-Since and server acknowledgement messages. For invalidation and polling-every-time algorithms, there will be only one-directional control messages transferred. This more or less 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 are going to explore them separately in our experiments.

In [CDF+98], the authors argue that in addition to high-level performance metrics such as the above-mentioned ones, we should also consider low-level details such as connection aborts and network delay because of heterogeneous bandwidth environment (e.g., modem pools connected to faster networks). In our research, we ignore the low level effect and focus mainly on the above metrics that will differentiate the performance of various cache consistency algorithms and their deployment locations.

We should point out that the overhead due to data collection for the performance metrics is negligible, because during the experiments we only record the timestamp at the beginning and the end of a Web interaction). Data is collected during the experiment, but all calculations are performed at the end of the measurement period. Thus, the only overhead is due to data recording, which is negligible. Similarly, cache statistic is also performed after all the Web interactions are complete.

Chapter 6

Experiments and Results

In order to compare the impact of different cache consistency algorithms on system performance, especially strong consistency algorithms, we conduct extensive experiments based on the performance metrics we listed in the previous chapter. Our results below are categorized by these metrics. For each set of experiments, we first get the boundary conditions, i.e., worst and best cases. The worst case, we believe, is when there is no cache, which gives us the lower bound of the performance. The upper bound is when the benchmark runs with proxy-side infinite cache. We don't have infinite space to implement a real infinite cache, but as long as there is no cache object removal during the whole 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. The machine that runs RBE and all the emulated browsers has only 128MB RAM. If we configure 100 EBs to run with, the size of browser cache for each EB cannot exceed 1MB. Real world caches are usually much bigger than 1MB. In our experiments, some browser cache becomes full several minutes into experiment. Therefore we gave up exploring deeper with client-side infinite cache.

6.1 WIPS - System Throughput

At the very beginning of our experiments, we wanted to see what the impact of client population will have 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 6.1 illustrates the change of WIPS as a result of change in client population.

We see that having 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 goes up 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 Web interactions per second. This is far more than what we got from 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. The network delay should not affect the WIPS unless it is the bottleneck.

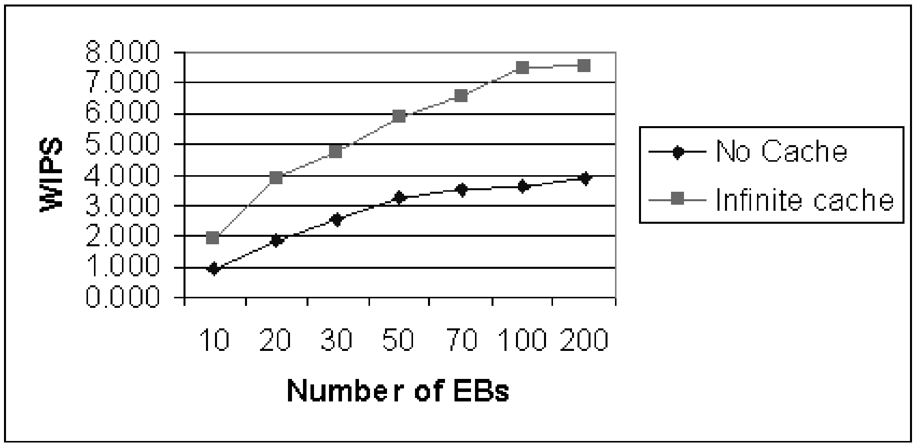

Figure 6.1: System Throughput vs Client Population (boundary conditions)

We believe that having the cache located at either client side or proxy server side will impact the WIPS differently, in addition to the choice of individual consistency algorithms. Therefore we implemented TTL, Invalidation (INV), Polling-every-time (POL) and Lease algorithms on both proxy cache and browser cache. Figure 6.2 is the throughput comparison of client-side deployment, and Figure 6.3 shows the result of proxy-side deployment. As we mentioned before, due to the main memory limitation, the client-side cache size cannot exceed 1MB. We started with 100KB and increased it by 100KB till 800KB. 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 champion. The reason why Lease does not perform as well as INV is that sometimes Lease expires before the objects actually get updated, which results in extra validation delay. Expecting that longer lease period might boost system throughput, we experiment the Lease algorithm with different lease periods. The result is that 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, 500KB seems to be the optimum for our system. We could see from Figure 6.2 that the WIPS decreases when cache size gets larger than 500KB. 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 fast enough, we don't 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 6.3 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 polling messages sent to server, which increases network

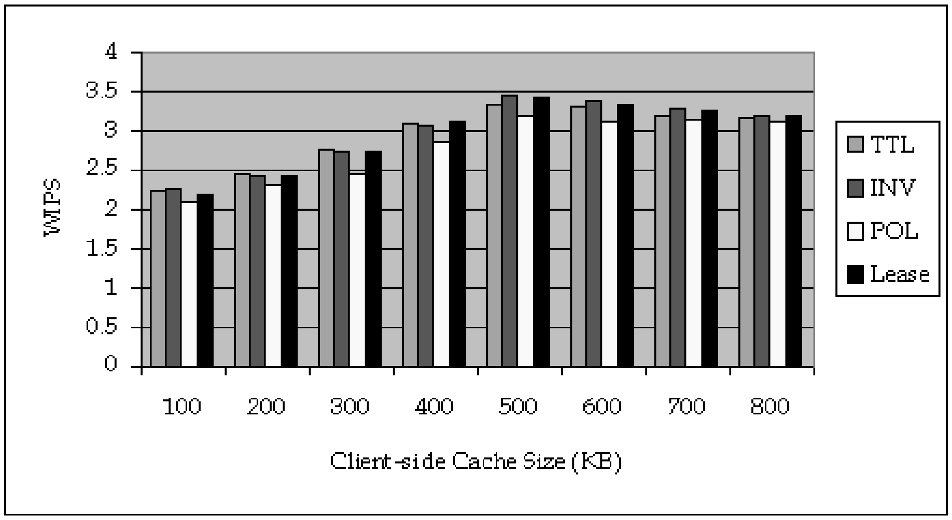

Figure 6.2: WIPS Comparison of client-side algorithms (fixing 100 EBs)

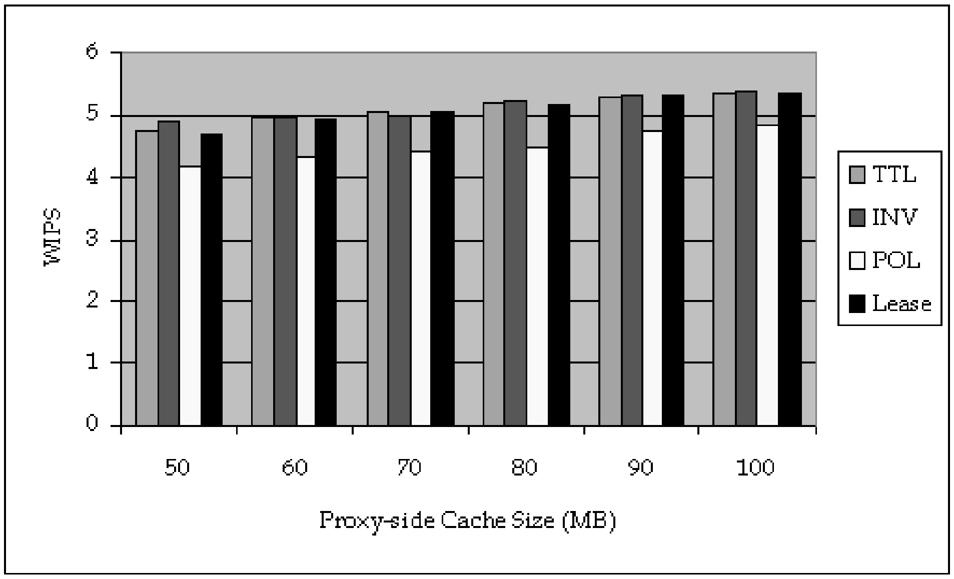

Figure 6.3: WIPS Comparison of proxy-side algorithms (fixing 100 EBs)

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.

After getting all the above results, we made a vertical comparison, i.e., for each consistency algorithm, we compared the WIPS results of client-side and proxy-side cache, and Figure 6.4 illustrates the comparison. For comparison purposes, since the cache size for each of the 100 EBs is 800KB in the case of browser cache, we chose the proxy-cache result when the cache size is 80MB.

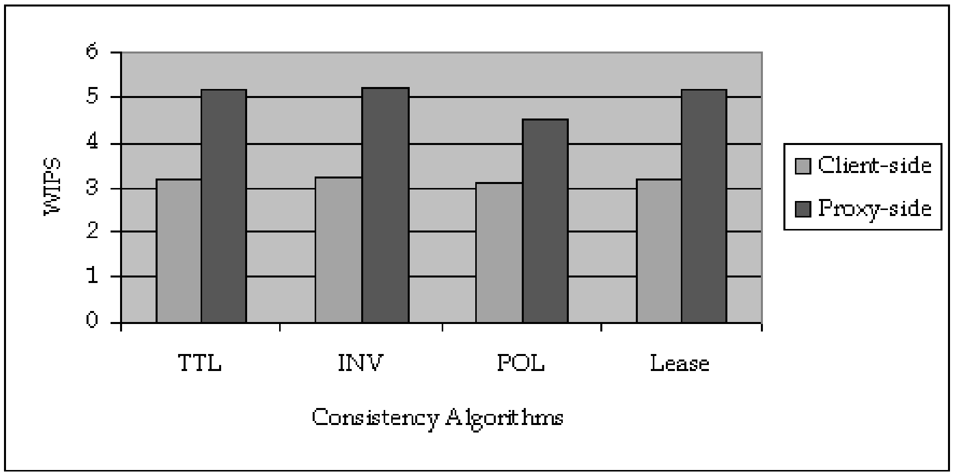

Figure 6.4: WIPS Comparison of consistency algorithms at different locations

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. At the same time, we believe that the limitation of our hardware configuration limits client-side caching performance significantly. We also notice that for client-side cache, these algorithms don't 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 size is bigger, therefore the impact of polling messages is reflected more significantly from the results.

6.2 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. The famous folkloric *8-second rule* is a hard evidence. If a Web site takes more than 8 seconds to deliver a page, the customers are very likely to leave the site. In this section, we present our experiment results on this critical success factor of e-commerce.

First of all, we want to see the comparison at boundary conditions. Fixing client population of 100 EBs, we ran the benchmark under different measurement intervals, from 1 to 6 hours. The benchmark was run first without cache, and then with proxy-side infinite cache. Figure 6.5 shows the results.

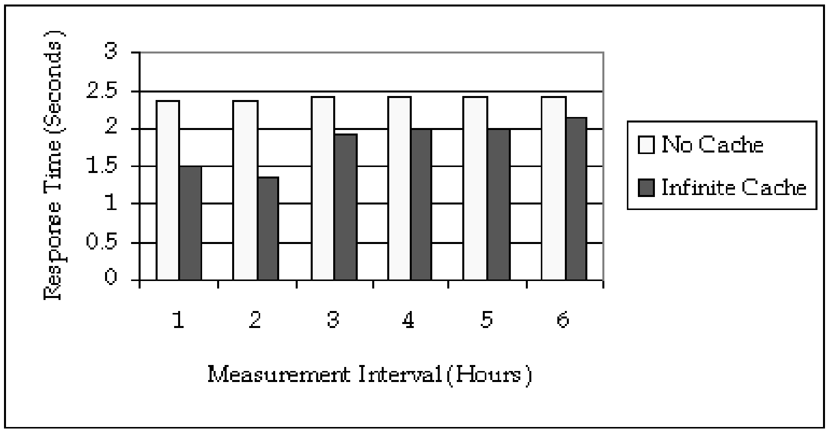

Figure 6.5: Response Time under boundary conditions

It seems that without cache, no matter how long the measurement interval is, the mean response time remains almost unchanged. This is because whenever a user request is submitted, it has to experience the network and application server delay. By contrast, we can see that 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 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, we ran the benchmark with client-side cache consistency algorithms, fixing cache size of each EB to be 500KB. Figure 6.6 is the result of the experiments. Then we ran the same set of experiments with proxy-side consistency algorithms, fixing proxy cache size to be 100MB. The experiment results are displayed in Figure 6.7. We notice that regardless of client population or cache location, INV performs a bit better than TTL and Lease (less 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.

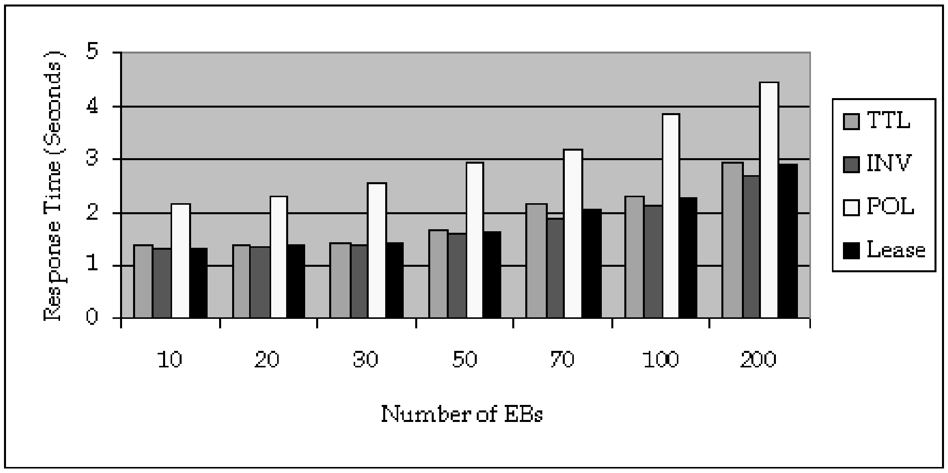

Figure 6.6: Response Time using client-side consistency algorithms

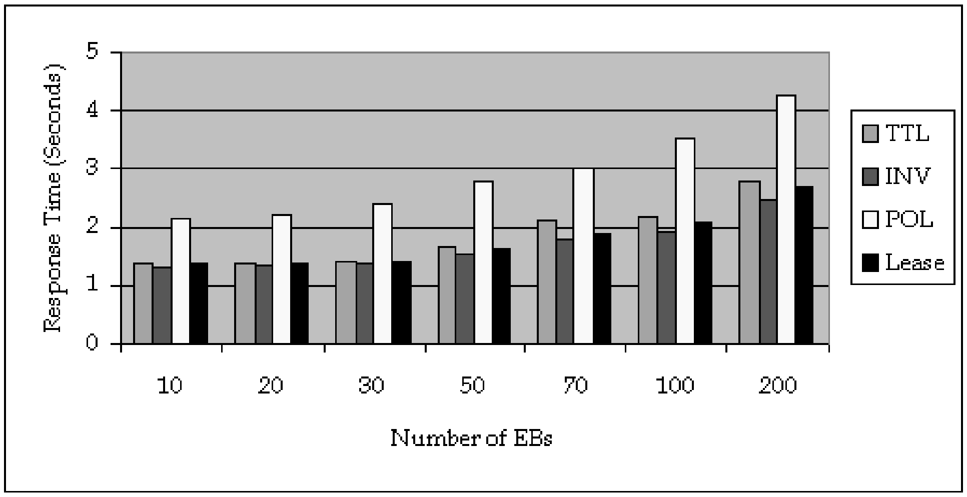

Figure 6.7: Response Time using proxy-side consistency algorithms

We also conducted a vertical comparison, comparing the response time difference of each consistency algorithm under both client-side and proxy-side caches. Figure 6.8 gives us the comparison result. We concluded that POL is the least favorable algorithm regardless of cache location. As we can see from the system throughput experiments, if the network transfer speed is slow, POL will perform even worse. For response time, we expect the same result.

6.3 Hit Ratio

Hit ratio is a performance metric for caching deployment. It does not make sense to measure the hit ratio if there is no cache at all. Therefore in our boundary case, we only ran the benchmark with proxy-side infinite cache. We first fixed the measurement interval of 1 hour, changed the client population from 10 to 200

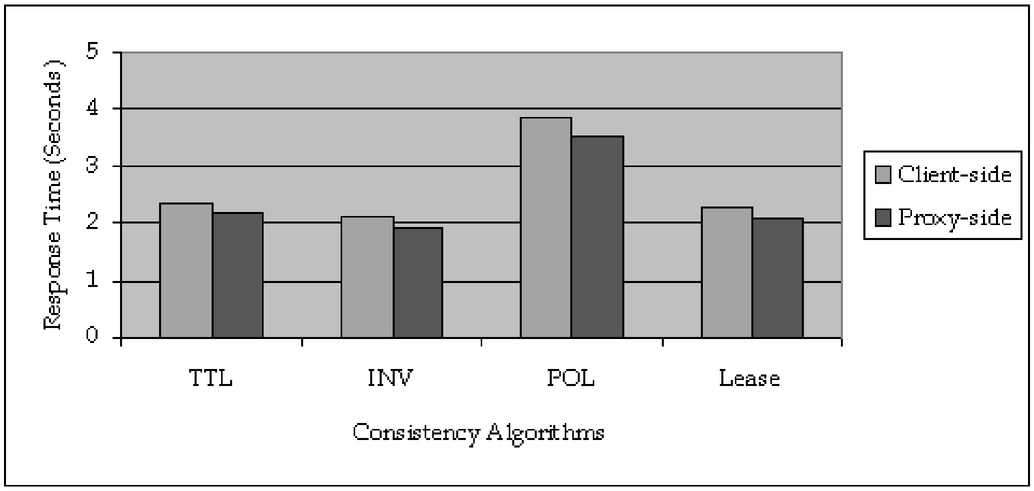

Figure 6.8: Response Time under different cache locations

concurrent EBs, and ran the benchmark with a proxy cache that has 100MB storage space. We found that 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 6.9 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.

We then fixed the measurement interval to be 1 hour, changed client population, and ran the benchmark with different consistency algorithms first with client-side cache, then with proxy-side. Figure 6.10 shows the change of cache hit ratio under client-side cache deployment, and Figure 6.11 gives the result of proxy-side deployment. From the results we did not see much difference, because the cache hits

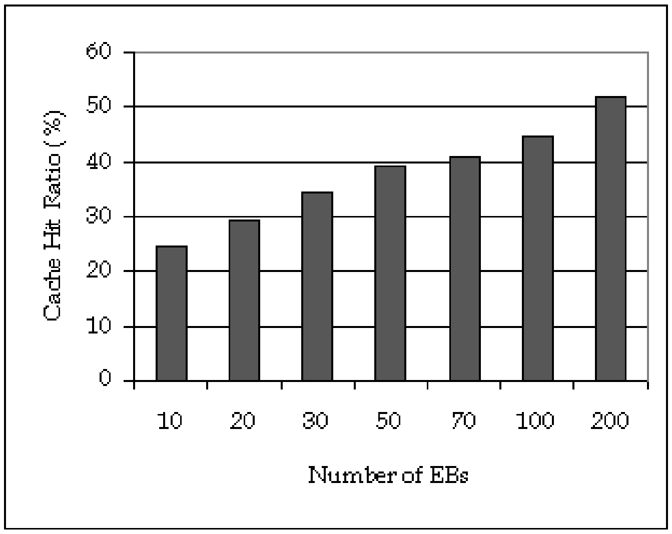

Figure 6.9: Cache hit ratio under proxy-side infinite cache

include both valid and stale hits. We analyzed the data and separated stale hits from valid ones, Figure 6.13 displays the results.

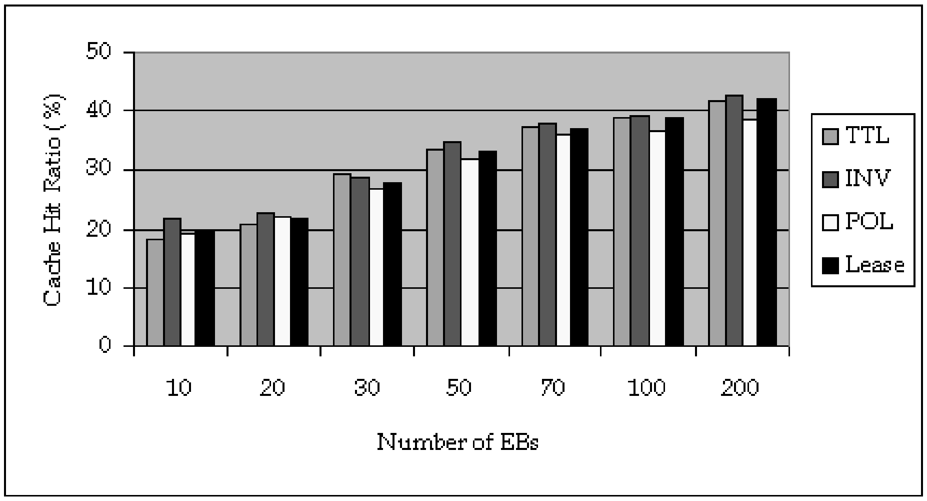

Figure 6.10: Comparison of client-side consistency algorithms

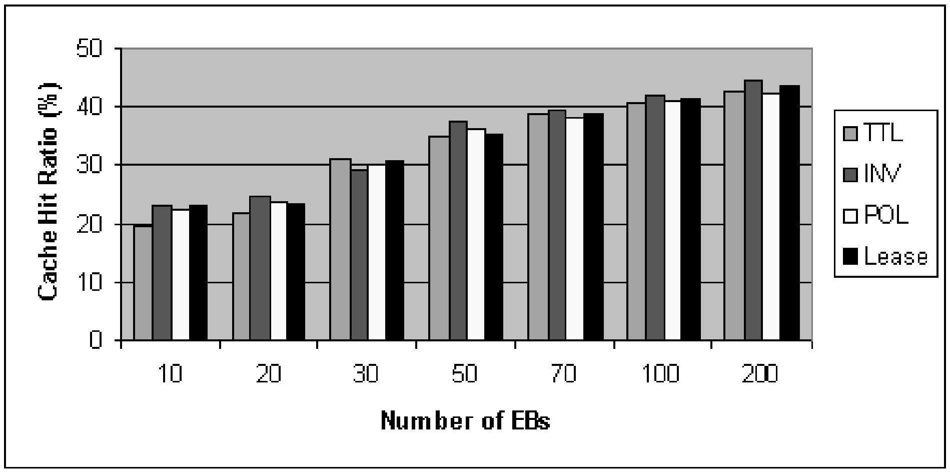

Figure 6.11: Comparison of proxy-side consistency algorithms

We extracted from the above experiment those data when the client population is 100 EBs. Then we compared the hit ratio under either client-side or proxy-side cache deployment. Figure 6.12 illustrates the results. Still, INV outperforms all other algorithms and achieved 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 6.13 shows the stale hit ratio while applying TTL and POL both at client and proxy side. We can see that the stale hit ratio is not very high in each case. This is because an online bookstore does not have as frequent data updates as other transaction systems do, therefore most objects remain valid

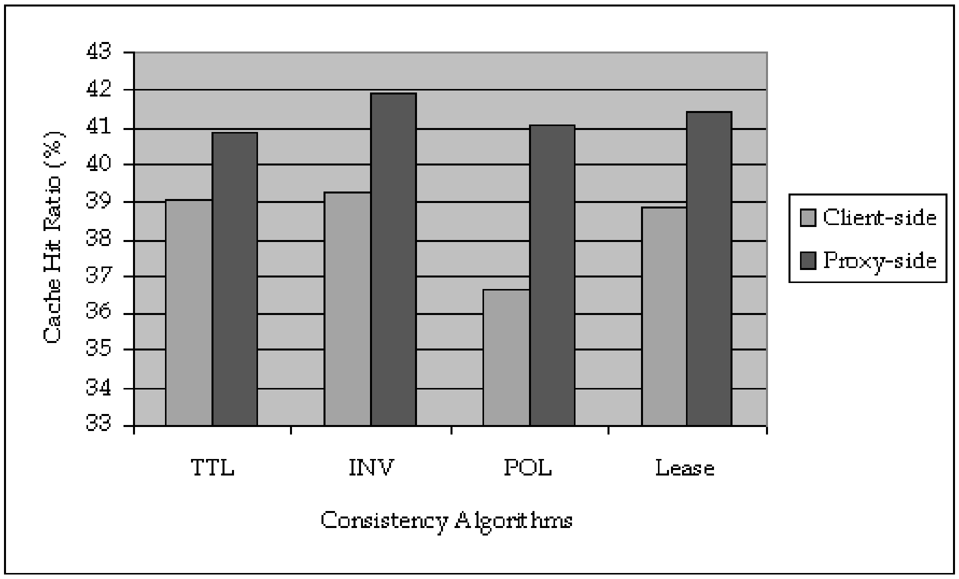

Figure 6.12: Cache hit ratio under different cache locations

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 don't often happen on stale objects.

6.4 Traffic

In order to measure the impact of consistency algorithms on network traffic, we made comparisons on two domains. First, 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 6.14 shows the total number of control messages under each algorithm, both at client side and proxy side. The

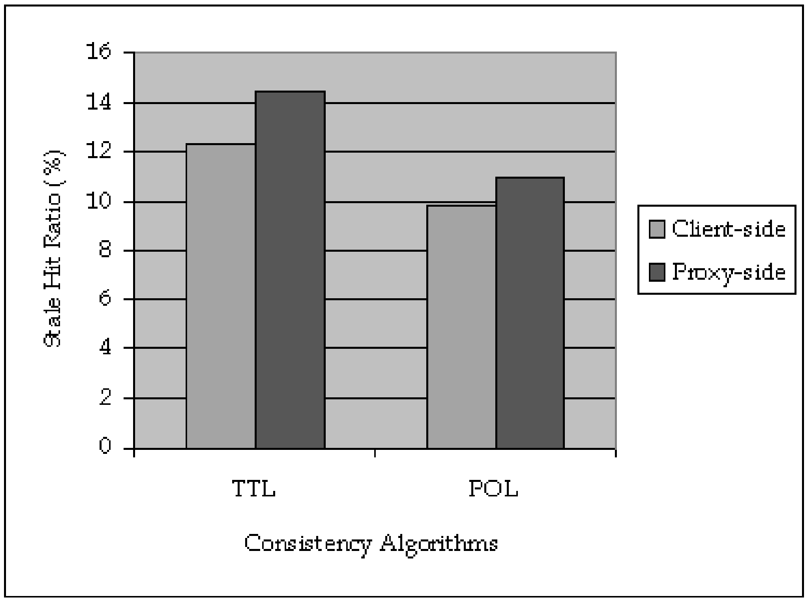

Figure 6.13: Stale hit ratio under different cache locations

measurement interval is fixed at one hour. We could see that POL generated a lot more control messages than other algorithms did. This creates extra delay experienced by end user.

Second, we tried to compare the total bytes of objects that users receive and the total bytes of objects that are served by cache. We fixed 100 emulated browsers as client population, 1 hour as the measurement interval, and set the cache at proxy-server side. We then ran the benchmark under infinite caching, TTL, INV, POL and Lease consistency algorithms. Figure 6.15 gives the experiment results.

We then conducted the same set of experiments with client-side cache. This time we don't have boundary case since we did not implement client-side infinite cache due to the main memory storage limitation. Figure 6.16 gives the experi-

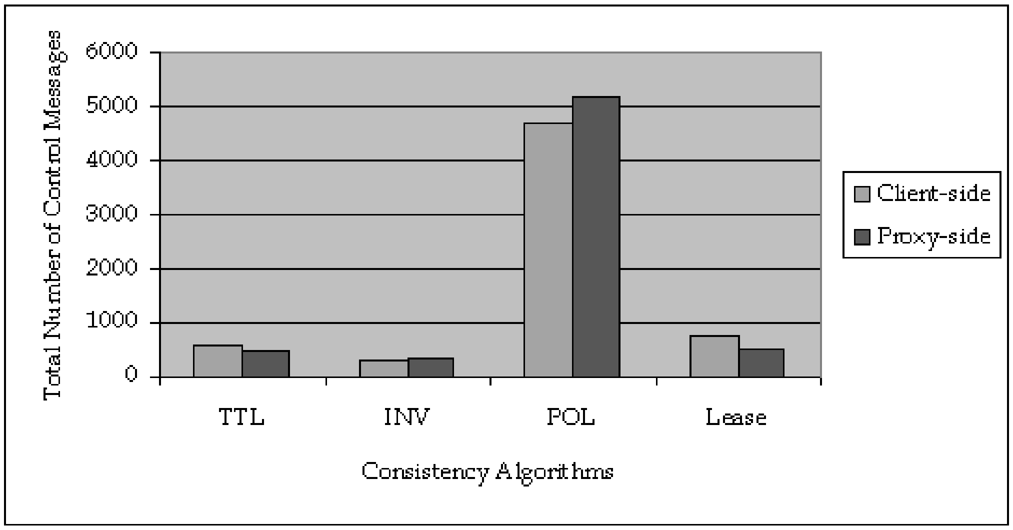

Figure 6.14: Control Message comparison using different consistency algorithms

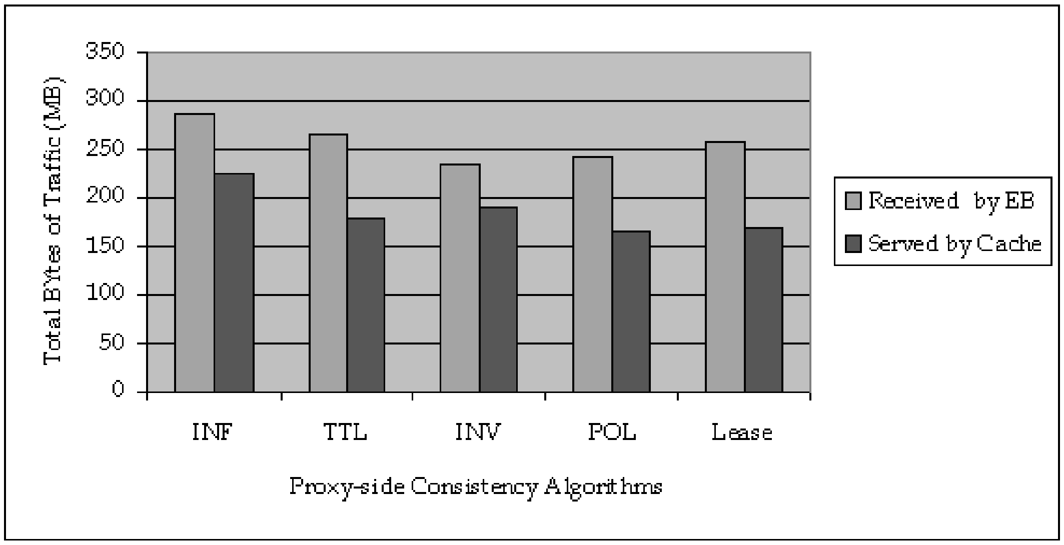

Figure 6.15: Traffic comparison under different proxy-side consistency algorithms

ment results. We don't see much difference on traffic savings for these algorithms, although it seems from the figure that Lease algorithm achieves the best. Our conclusion is that, 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.

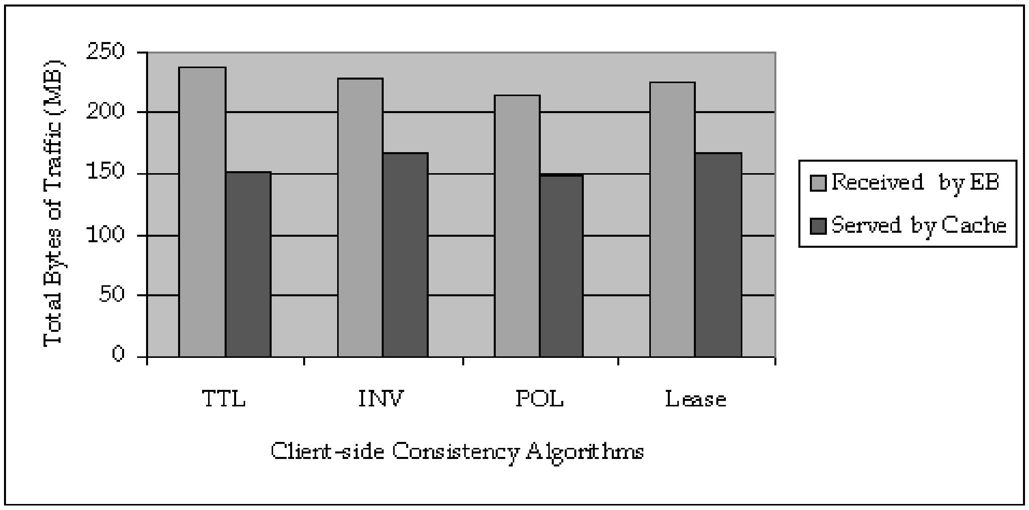

Figure 6.16: Traffic comparison under different client-side consistency algorithms

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 other 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. Therefore, this set of results are discarded.

Chapter 7

Conclusion and Future Work

7.1 Contributions

We identify following aspects as the contributions of our research.

First of all, we use TPC-W benchmark, the industry standard for measuring e-commerce activity, as our testbed. The advantage of this is that our research is based on a representative workload that represents online business behavior. At the same time, the benchmark implementation that we use is a third-party package and it is available online. This makes our experiment results convenient for comparison.

Secondly, we studied most of the cache consistency algorithms in the literature, classifying them into a two-dimension taxonomy. **The basis of our classification is both on the level of consistency that an algorithm could enforce, and who acts the major role in the consistency control process**. Focusing on strong consistency, we picked up one representative algorithm in each of the 3 categories that enforce strong cache consistency. We implemented these three strong consistency algorithms, as well as TTL, the most popular weak consistency algorithm. We conducted extensive experiments and compared the performance of these algorithms. Our experiment results, no matter which algorithm they favour, will be interesting to both academic and commercial parties.

Thirdly, we chose as many as four performance metrics in evaluating consistency algorithms. This is rare in the literature, as most research papers focus only on one particular metric, e.g., response time or hit ratio. In addition to response time, hit ratio and total bytes of messages, which are traditional performance indicators for cache study, we also use WIPS, a performance metric of TPC-W benchmark itself, as a measurement of system throughput to compare consistency algorithms. Therefore, our experiment results are attractive to commercial vendors.

We also discussed the choice of cache location and compared the performance results of each consistency algorithm at both client side and proxy side. This gives us a complete picture of the advantages and disadvantages of cache deployment and decision on consistency control, which could be valuable to many online e-business Web sites who are considering caching mechanism for their performance enhancement. We have already mentioned in Chapter 1 the misunderstanding that Web caching systems should apply weak consistency instead of strong, because the former requires less response time and server load. Our experiments prove that from any aspect of performance measurement, weak algorithms don't necessarily outperform strong ones. By analyzing the experiment results, we also found that in the category of strong cache consistency, event-driven algorithms perform better than time-based, due to the nature of e-business, i.e., it is unpredictable when customers will come and when the transaction will be made.

We draw our conclusion (in Section 7.2) based on objective analysis of experiment data. The impact of cache size and network delay on the overall caching performance are studied and the result is of reference value to the real world cache deployment.

7.2 Conclusion and Future Work

We have analyzed and compared the performance of three strong cache consistency control algorithms: Invalidation, Polling-every-time and Lease. As a comparison base, we also implemented TTL algorithm, the popular consistency mechanism that enforces weak consistency. We compared the pros and cons of deploying these mechanisms either at client or proxy side. Our experiment results show that proxy side cache improves system performance better than client side cache does, and INV performs the best regardless of cache location. POL is the least favorable algorithm from our results. It has the longest response time, producing the most message traffic. On the other hand, we are aware that POL is the easiest algorithm to implement.

The advantage of Invalidation algorithm over TTL and Polling-every-time is twofold. First, as we can see from our experiments, Invalidation has the fewest control message transfer, shortest response time, similar throughput as TTL while keeping cached objects fresh. On the other hand, comparing to TTL, a time-based consistency algorithm, 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. Instead, 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 we expected at the beginning of our experiments, the performance of proxy-side cache beats that of client-side cache significantly. From storage point of view, having the cache shared by a group of users reduces redundant object storage. From consistency point of view, it is easier to maintain consistency between server and one proxy cache rather than multiple browser caches. As mentioned in the previous chapter, however, the hardware limitation of our experiments affects the client-side cache performance. According to our experiment results, browser cache is not recommended compared to proxy cache even the storage space is not an issue. We believe that having both browser and proxy cache in place will result in better performance than applying either one alone, but we did not explore the combination in this thesis.

The performance result of our experiments may not be optimum, partly because we chose the simplest and generally accepted LRU algorithm as our cache replacement algorithm. We did make some enhancement 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[CI97]. These factors include document download time delay, network travelling costs of different Web objects, etc. However, our results are still comparable to each other because each caching algorithm is deployed under the same replacement algorithm, the network environment and Web page set is the same, too.

The operating system that we use is another limitation of our research. We are not sure yet whether the long file search delay on the hard disk is due to the Windows 2000 itself, but because of that, we had to choose store size of 10,000 items instead of 100,000, the one representative of a mid-size online bookstore.

Due to hardware limitations, we only considered one proxy cache in our experiments. This prevented us from examining the performance of different cache consistency algorithms under a cooperative caching architecture. For large scale organizations who have thousands of end users possibly locating at geographically separated sites, the use of a single proxy cache by all users will certainly not achieve desirable system performance. We plan to add more hardware to our experiment network environment and test the performance of the examined algorithms again and analyze the impact of proxy cache cooperation on individual consistency control algorithms.

A lot of future work remains ahead. We plan to move the benchmark to Solaris or Linux, and change the store size to 100,000, then run the whole set of experiments over again. In this way, we can tell how the consistency control algorithms perform in a mid-size online bookstore scenario. We can also tell whether the slow file search issue we encountered in our experiment is indeed due to Windows 2000 operating system. We are going to increase the processing power of both client and server machine, and see whether our current conclusions remain the same. All the performance expectation and estimates need to be checked against real experiment results.

In our current experiment settings, all the EBs are managed by one single RBE.

To run the benchmark under a cooperative caching architecture, we could have multiple RBEs in the system, each RBE managing a group of EBs. The result of this configuration will be interesting.

As the future work, a set of stress tests could be conducted on the current environment to evaluate the performance of these consistency algorithms. We could run the benchmark with extremely many number of EBs (e.g., 1000) or adjust the network delay parameter to be very long to see whether the results of relative performance will be different.

With regard to cache consistency algorithm, we could explore further with piggybacking mechanism in cache consistency control and try to come up with an algorithm that requires the interaction between server and client but enforces stronger cache consistency.

Bibliography

[AC98] J. Almeida and P. Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. *Computer Networks and ISDN Systems*, 30(22–23):2179–2192, 1998.

[ASA+95] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching Proxies: Limitations and Potentials. In *Proceedings of the 4th International World Wide Web Conference*, Boston, MA, December 1995.

[BDR97] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings USENIX'97*, pages 289–303, 1997.

[Cat92] Vincent Cate. Alex – A Global File System. In *Proceedings of the USENIX File System Workshop*, pages 1–11, Ann Arbor, Michigan, 1992.

[CDF+98] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web Proxy Caching: The Devil is in the Details. *1998 Workshop on Internet Server Performance*, June 1998.

[CDI99] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, 1999.

[CDN+96] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, pages 153–164, San Diego, CA, January 1996.

[CI97] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[CL98] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World Wide Web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[CZB98] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedinds of Middleware'98*, pages 289–303, 1998.

[DHS93] P.B. Danzig, R.S. Hall, and M.F. Schwartz. A Case for Caching File Objects Inside Internetworks. In *Proceedings of the SIGCOMM '93 conference*, pages 239–248, San Francisco, CA, 1993.

[DR97] F. Douglis and M. Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proceedings of the USENIX*

Symposium on Internet Technologies and Systems, Monterey, CA, December 1997.

[EBH$^+$01] H.K. Edwards, M.A. Bauer, H.L., 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*, pages 148–157, April 2001.

[FCAB00] L. Fan, P. Cao, J. Almeida, and A.Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[FCD$^+$99] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. In *Proceedings of IEEE INFOCOM*, pages 107–116, New York, NY, March 1999.

[GC89] 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*, pages 202–210, 1989.

[GS96] J. Gwertzman and M. Seltzer. World Wide Web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference*, pages 141–152, San Diego, CA, January 1996.

[JB00] S. Jin and A. Bestavros. GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams. Technical Report 2000-011, Boston University, April 2000.

[KLM97] T.M. Kroeger, D.E. Long, and J.C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[KW97] 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*, pages 1–12, Monterey, CA, December 1997.

[KW98] B. Krishnamurthy and C.E. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. *Computer Networks and ISDN Systems*, 30(1–7):185–193, 1998.

[LN01] Q. Luo and J.F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. In *Proceedings of 27th VLDB Conference*, Rome, Italy, September 2001.

[Luo98] Ari Luotonen. *Web Proxy Servers.* Prentice Hall PTR, 1998.

[ÖV99] M. Tamer Özsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd edition.* Prentice Hall PTR, 1999.

[PH97] D. Povey and J. Harrison. A Distributed Internet Cache. In *Proceedings of the 20th Australasian Computer Science Conference*, February 1997.

[PHG96] D.A. Patterson, J.L. Hennessy, and D. Goldberg. *Computer Architecture: A Quantitative Approach, 2nd edition.* Morgan Kaufmann Publishers Inc., 1996.

[PP01] K. Psounis and B. Prabhakar. A Randomized Web-Cache Replacement Scheme. In *Proceedings of IEEE INFOCOM 2001*, pages 1407–1415, Anchorage, AK, April 2001.

[RS02] M. Rabinovich and O. Spatscheck. *Web Caching and Replication.* Addison-Wesley, 2002.

[RSB99] P. Rodriguez, C. Spanner, and E.W. Biersack. Web Caching Architectures: Hierarchical and Distributed Caching. In *Proceedings of the 4th International Web Caching Workshop*, San Diego, April 1999.

[RV00] L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[RW98] A. Rousskov and D. Wessels. Cache Digests. *Computer Networks and ISDN Systems*, 30(22–23):2155–2168, 1998.

[SAYZ99] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploring Result Equivalence in Caching Dynamic Web Content. In *Proceedings of USITS'99: the 2nd USENIX Symposium on Internet Technologies & Systems*, Boulder, Colorado, October 1999.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of*

Summer USENIX Technical Conference, pages 119–130, Portland, OR, June 1985.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. Technical Report TR-819, MIT, March 2001.

[TDVK99] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 273–284, Austin, TX, June 1999.

[Tra01] Transaction Processing Performance Council (TPC). TPC Benchmark^{TM} W (Web Commerce) Specification Version 1.4. February 2001. http://www.tpc.org/tpcw.

[VR98] V. Valloppillil and K. Ross. Cache Array Routing Protocol v1.1. Internet Draft, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnproxy/html/carp.asp, February 1998.

[WA97] R. Wooster and M. Abrams. Proxy caching that Estimates Page Load Delays. In *Proceedings of the 6th International World Wide Web Conference*, pages 325–334, Santa Clara, CA, April 1997.

[WAS+96] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Procedings of the ACM SIGCOMM'96 Conference*, Montreal, Canada, June 1996.

[WC97a] D. Wessels and K. Claffy. Application of Internet Cache Protocol (ICP), version 2. Request for Comments: 2187, September 1997.

[WC97b] D. Wessels and K. Claffy. Internet Cache Protocol, Version 2. Request for Comments: 2186, September 1997.

[Wes95] Duane Wessels. Intelligent Caching for World-Wide Web Objects, 1995.

[WVS+99] A. Wolman, G.M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H.M. Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of SOSP'99: 17th ACM Symposium on Operating Systems Principles*, pages 16–31, Kiawah Island Resort, near Charleston, SC, December 1999.

[YADL98] 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*, pages 285–294, 1998.

[YADL99] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases for consistency in large-scale systems. *Knowledge and Data Engineering*, 11(4):563–576, 1999.

[YBS99] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency

Architecture. In *Proceedings of the ACM SIGCOMM'99*, pages 163–174, Boston, MA, September 1999.

[YFIV00] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. In *Proceedings of the 26th VLDB Conference*, pages 188–199, Cairo, Eygpt, September 2000.

[ZFJ97] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching. In *Proceedings of the NLANR Web Cache Workshop*, June 1997.

Appendix A

Glossary

CARP – Cache Array Routing Protocol

CGI – Common Gateway Interface

DCCP – Dynamic Content Caching Protocol

DNS – Domain Naming System

EB – Emulated Browser

FIFO – First In First Out

HTTP – HyperText Transfer Protocol

ICP – Internet Cache Protocol

INV – Invalidation, a strong consistency control algorithm

IMS – If-Modified-Since

ISP – Internet Service Provider

JSP – Java Server Pages

LFU – Least Frequently used cache replacement algorithm

LRU – Least Recently Used cache replacement algorithm

LRV – Lowest Relative Value cache replacement algorithm

NSFNET – National Science Foundation Network

OCLC – Online Computer Library Center

PCV – Piggyback Cache Validation, a weak consistency control algorithm

POL – Polling-every-time, a strong consistency control algorithm

PSI – Piggyback Server Invalidation, a weak consistency control algorithm

RBE – Remote Browser Emulator

SUT – System Under Test

TCP – Transmission Control Protocol

TPC – Transaction Processing Performance Council

TPC-W – TPC Benchmark^{TM} W, a transactional Web benchmark

TTL – Time-To-Live, one of the representative weak consistency control algorithms

URL – Uniform Resource Locator

WIPS – Web Interaction Per Second, one of the major performance metrics for the benchmark

W3C – World Wide Web Consortium

Appendix B

TPC-W Database Entity Definition

```
CREATE TABLE CUSTOMER
         ( C_ID           INT not null,      //unique ID per customer
           C_UNAME        VARCHAR(20),       //unique user name for customer
           C_PASSWD       VARCHAR(20),       //user password for customer
           C_FNAME        VARCHAR(17),       //first name of customer
           C_LNAME        VARCHAR(17),       //last name of cusomer
           C_ADDR_ID      INT,               //address ID of customer
           C_PHONE        VARCHAR(18),       //phone number of customer
           C_EMAIL        VARCHAR(50),       //email address of customer
           C_SINCE        DATE,              //date of customer registration
           C_LAST_LOGIN   DATE,              //date of last visit
           C_LOGIN        TIMESTAMP,         //start of current customer session
           C_EXPIRATION   TIMESTAMP,         //current customer session expiry
           C_DISCOUNT     REAL,              //percentage discount for customer
           C_BALANCE      DOUBLE,            //balance of customer
           C_YTD_PMT      DOUBLE,            //Year-To-Date payment
           C_BIRTHDATE    DATE,              //birth date of customer
           C_DATA         VARCHAR(510),      //miscellaneous information
           PRIMARY KEY(C_ID))
```

Figure B.1: DDL to create TPC-W database master tables

```
CREATE TABLE COUNTRY
                  ( CO_ID               INT not null,         //unique country ID
                   CO_NAME              VARCHAR(50),          //name of country
                   CO_EXCHANGE          DOUBLE,               //exchange rate to US Dollars
                   CO_CURRENCY          VARCHAR(18),          //name of currency
                   PRIMARY KEY(CO_ID))

CREATE TABLE ADDRESS
                  ( ADDR_ID             INT not null,         //unique address ID
                   ADDR_STREET1         VARCHAR(40),          //street address, line 1
                   ADDR_STREET2         VARCHAR(40),          //street address, line 2
                   ADDR_CITY            VARCHAR(30),          //name of city
                   ADDR_STATE           VARCHAR(20),          //name of state
                   ADDR_ZIP             VARCHAR(10),          //zip code or postal code
                   ADDR_CO_ID           INT,                  //unique ID of country
                   PRIMARY KEY(ADDR_ID))

CREATE TABLE ORDERS
                  ( O_ID                INT not null,         //unique ID per order
                   O_C_ID               INT,                  //customer ID of order
                   O_DATE               DATE,                 //order date and time
                   O_SUB_TOTAL          DOUBLE,               //subtotal of all order-line items
                   O_TAX                DOUBLE,               //tax over the subtotal
                   O_TOTAL              DOUBLE,               //total for this order
                   O_SHIP_TYPE          VARCHAR(10),          //method of delivery
                   O_SHIP_DATE          DATE,                 //order ship date
                   O_BILL_ADDR_ID            INT,             //address ID to bill
                   O_SHIP_ADDR_ID            INT,             //address  ID to ship order
                   O_STATUS             VARCHAR(15),          //order status
                   PRIMARY KEY(O_ID))

CREATE TABLE CC_XACTS
                  ( CX_O_ID             INT not null,         //unique order ID (O_ID)
                   CX_TYPE              VARCHAR(10),          //credit card type
                   CX_NUM               VARCHAR(20),          //credit card number
                   CX_NAME              VARCHAR(30),          //name on credit card
                   CX_EXPIRE            DATE,                 //expiration date of credit card
                   CX_AUTH_ID           CHAR(15),             //authorization for trans. amount
                   CX_XACT_AMT          DOUBLE,               //amount for this transaction
                   CX_XACT_DATE         DATE,                 //date and time of authorization
                   CX_CO_ID             INT,                  //country where trans. originated
                   PRIMARY KEY(CX_O_ID))
```

Figure B.2: DDL to create TPC-W database master tables (cont'd)

```
CREATE TABLE ITEM
            ( I_ID             INT not null,      //unique ID of item
              I_TITLE          VARCHAR(60),       //title of item
              I_A_ID           INT,               //author ID of item
              I_PUB_DATE       DATE,              //date of release of the product
              I_PUBLISHER      VARCHAR(60),       //publisher of item
              I_SUBJECT        VARCHAR(60),       //subject of book
              I_DESC           VARCHAR(500),      //description of Item
              I_RELATED1       INT,               //related item 1
              I_RELATED2       INT,               //related item 2
              I_RELATED3       INT,               //related item 3
              I_RELATED4       INT,               //related item 4
              I_RELATED5       INT,               //related item 5
              I_THUMBNAIL      VARCHAR(40),       //pointer to thumbnail image
              I_IMAGE          VARCHAR(40),       //pointer to image
              I_SRP            DOUBLE,            //suggested retail price
              I_COST           DOUBLE,            //cost of item
              I_AVAIL          DATE,              //when item is available
              I_STOCK          INT,               //quantity in stock
              I_ISBN           CHAR(13),          //product ISBN
              I_PAGE           INT,               //number of pages of book
              I_BACKING        VARCHAR(15),       //type of book, paper or hard back
              I_DIMENSIONS     VARCHAR(25),       //size of book in inches
              PRIMARY KEY(I_ID))

CREATE TABLE ORDER_LINE
            ( OL_ID            INT not null,      //unique order line item ID
              OL_O_ID          INT not null,      //order ID of order line
              OL_I_ID          INT,               //unique item ID (I_ID)
              OL_QTY           INT,               //quantity of Item
              OL_DISCOUNT      DOUBLE,            //Percentage discount off of I_SRP
              OL_COMMENTS      VARCHAR(110),      //special instructions
              PRIMARY KEY(OL_ID, OL_O_ID))

CREATE TABLE AUTHOR
            ( A_ID             INT not null,      //unique author ID
              A_FNAME          VARCHAR(20),       //first name of author
              A_LNAME          VARCHAR(20),       //last name of author
              A_MNAME          VARCHAR(20),       //middle name of author
              A_DOB            DATE,              //date of birth of author
              A_BIO            VARCHAR(500),      //about the author
              PRIMARY KEY(A_ID))
```

Figure B.3: DDL to create TPC-W database master tables (cont'd)