The performance of a client-side web caching system

A Thesis

 Presented to

the Faculty of

California Polytechnic State University, San Luis Obispo

In partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ying-Lin Chen

April 2000
AUTHORIZATION FOR REPRODUCTION
OF MASTER’S THESIS

I grant permission for the reproduction of this thesis in its entirety or any of its parts, without further authorization from me.

_____________________________________________
Signature

_____________________________________________
Date
TITLE: The performance of a client-side web caching system

AUTHOR: Ying-Lin Chen

DATE SUBMITTED: April 28, 2000

Dr. Mei-Ling Liu
Adviser or Committee Chair
Signature

Dr. Hugh Smith
Committee Member
Signature

Dr. Lois Brady
Committee Member
Signature
ABSTRACT

The performance of a client-side web caching system

By Ying-Lin Chen

The World Wide Web has grown exponentially in the past few years. Consequently, there is an ever-increasing demand for network bandwidth. One way to optimize network bandwidth usage is to implement a caching system which stores previously fetched files at a location close to the web user, thereby allowing the user to experience reduced response time, and also potentially allowing network bandwidth to be conserved.

This thesis investigates the implementation of a web caching system located on the client system, and studies the impact of such a system on the request latency and data throughput. The study also compares the performance of such a system implemented on two different computer platforms.

Our test data show that client-side caching reduces the user-perceived latency. The latency-reduction is more significant when the request is for small files than for large files. However, the benefit of client-side caching levels off when the size of the requested file becomes very large.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Caching</td>
<td>4</td>
</tr>
<tr>
<td>Caching Policies</td>
<td>5</td>
</tr>
<tr>
<td>The Internet Infrastructure</td>
<td>6</td>
</tr>
<tr>
<td>HTTP Elements</td>
<td>6</td>
</tr>
<tr>
<td>Outline of the Thesis</td>
<td>8</td>
</tr>
<tr>
<td>Chapter 2. Related Works</td>
<td>10</td>
</tr>
<tr>
<td>Cache Manager</td>
<td>10</td>
</tr>
<tr>
<td>Squid Proxy Performance</td>
<td>11</td>
</tr>
<tr>
<td>Delta-Encoding Technique</td>
<td>12</td>
</tr>
<tr>
<td>Chapter 3. Motivation and Implementation Scheme</td>
<td>15</td>
</tr>
<tr>
<td>Client Systems</td>
<td>15</td>
</tr>
<tr>
<td>Software Modules</td>
<td>16</td>
</tr>
<tr>
<td>Caching Policy</td>
<td>19</td>
</tr>
<tr>
<td>Client-Server Geography</td>
<td>20</td>
</tr>
<tr>
<td>Chapter 4. Performance Results</td>
<td>21</td>
</tr>
<tr>
<td>Test Data Collection</td>
<td>21</td>
</tr>
<tr>
<td>Terminology</td>
<td>22</td>
</tr>
<tr>
<td>Celeron 266 Performance Analysis</td>
<td>24</td>
</tr>
<tr>
<td>Pentium III 500 Performance Analysis</td>
<td>28</td>
</tr>
<tr>
<td>Chapter 5. Conclusion and Summary</td>
<td>34</td>
</tr>
<tr>
<td>Celeron vs. Pentium III</td>
<td>34</td>
</tr>
<tr>
<td>Conclusion &amp; Summary</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 6. Future Work ................................................................. 40
Appendix ................................................................. 42
  Browser Caching Configuration ........................................ 42
  Browser Port configuration ........................................... 42
  Simulation Software: ...................................................... 42
Bibliography ................................................................. 68
LIST OF FIGURES

Figure 1: The network traffic volume (without caching implemented)………………………………….. 3
Figure 2: The network traffic volume (with caching implemented)……………………………………... 4
Figure 3: The communication between the client-side web caching components……………………….. 16
Figure 4: The flow chart of the software modules communication time frame…………………………….. 18
Figure 5: Latency measurement of small set ASCII data files - Celeron 266MHz, 96MB RAM……….. 24
Figure 6: Latency measurement of large set ASCII data files - Celeron 266MHz, 96MB RAM………... 24
Figure 7: Throughput vs. file size - Celeron 266MHz, 96MB RAM…………………………………….. 27
Figure 8: Latency measurement of small set ASCII data files - Pentium III 500MHz, 256MB RAM….. 29
Figure 9: Latency measurement of large set ASCII data files - Pentium III 500MHz, 256MB RAM….. 29
Figure 10: Throughput vs. file size - Pentium III 500MHz, 256MB RAM……………………………… 32
Figure 11: Socket latency comparison of small set ASCII data files - Celeron vs. Pentium III…………. 34
Figure 12: Socket latency comparison of large set ASCII data files - Celeron vs. Pentium III…………. 34
Figure 13: Browser latency comparison of small set ASCII data files - Celeron vs. Pentium III……….. 35
Figure 14: Browser latency comparison of large set ASCII data files - Celeron vs. Pentium III………... 35
Figure 15: Socket throughput comparison - Celeron vs. Pentium III……………………………………. 37
LIST OF TABLES

Table 1: Socket latency measurement - Celeron 266MHz, 96MB RAM…………………………………26
Table 2: Browser latency measurement - Celeron 266MHz, 96MB RAM……………………………….26
Table 3: Throughput data - Celeron 266MHz, 96MB RAM……………………………………………...28
Table 4: Socket latency measurement - Pentium III 500MHz, 256MB RAM……………………………31
Table 5: Browser latency measurement - Pentium III 500MHz, 256MB RAM…………………………. 31
Table 6: Throughput data - Pentium 500MHz, 256MB RAM…………………………………………… 33
CHAPTER 1: Introduction

The World Wide Web, sometimes called WWW or the Web, combines the client-server handshaking and communications protocol, the Hypertext Transfer Protocol (HTTP), with a document formatting language, the Hypertext Makeup Language (HTML). The WWW has gained attention dramatically in the past few years. Statistics from the NSFNET backbone show the WWW traffic surpassing File Transfer Protocol (FTP), Telnet, and Gopher usage in just six months, from December 1994 to April 1995 [8, 12]. The exponential growth of the WWW has caused serious performance degradation: it has dramatically increased the Internet traffic and saturated the network bandwidth, as well as significantly prolonged in user-perceived latency while accessing web pages. Currently HTTP/1.0 is widely adapted as the Internet transmission protocol skeleton. Every web page, and the embedded images it contains, need to establish a separate Transfer Control Protocol (TCP) connection to handle its own transaction. TCP use the algorithm called Slow-Start to determines the best rate of transmission in the particular network path bandwidth, and this process takes time to configure itself into the steady state. Ideally, the steady state is the rate of number of packets leaving the server equal to the rate of number of packets reaching the client end [3, 5].

According to [1], [2], [3], [5], [8], [9], and [15], the majority of web document sizes are within 100-100K bytes; less than 10% web documents sizes are larger than 100K bytes. The mean HTML document size is 10K bytes, the icon is 2K bytes, the image is 20K bytes, and the photo is 200K bytes. The majority of web documents that flow through the Internet only need a relatively small number of data packets to accomplish downloading/uploading actions. Consequently, the TCP connections associated with these packets terminate before they have a chance to settle into the steady state, best transmission rate [6]. In an attempt to alleviate this problem, HTTP version 1.1 was proposed as a means for reducing both document transfer latency and network traffic. The HTTP/1.1 performance enhancement is based on the assumption that the network’s bandwidth is the bottleneck of WWW traffic. Particularly, HTTP/1.1
proposes several protocol enhancement schemes, such as persistent connections (referred to as PHTTP), pipelining, link level document compression, and caching [10].

Prior to persistent connections, each Uniform Resource Locator (URL) request establishes a separate TCP connection to fetch the data. This increased the load on HTTP servers and caused network packet congestion, and contributed to the degeneration of the user-perceived latency. Persistent connections use a single TCP connection to communicate between a client / server pair. This reduces network congestion by reducing the number packets generated in response to TCP connection requests, and gives the TCP connections sufficient time to reach a steady-state transmission rate [10, 11].

Pipelining in HTTP/1.1 lets a client send multiple requests to a server without waiting for a response to each request. Furthermore, each packet can contain multiple requests or replies. This is done by placing a separator character between the requests / replies within the packets. These features let the client and server send packets that are as full as possible, thereby reducing the total number of request and response packets [10, 11].

Link level document compression such as the delta encoding scheme where the Internet server only sends the difference – i.e., the “delta” – between the latest version of the web page and the cached version of the web page on the remote client [17]. In other words, server does not send the entire web page content; it only sends part of it. Thus the delta encoding technique saves network bandwidth and reduces the user-perceived latency. However, the server needs time to process and compress the delta. Likewise, the client needs time to decompress the delta and combine it with its cached version in order to generate up-to-date web content. At a point, the extra processing time can overwhelm the latency time saving benefits. The HTTP/1.1 protocol also supports application level compression by implementing the tag “Content-Encoding”. The “Content-Encoding” tag value indicates what the compression method is applied on the message content; therefore the corresponding decoding procedure must be applied on the message data in order to obtain the media-type referenced by the Content-Type header field [11].

Caching has long been used in the field of Computer Science. Operating systems and databases commonly use caching to improve performance by storing data that are frequently accessed. The initial implementation of caching technology in the network is a means of storing web documents from the original web server closer to the client. By reducing the fetching distance between the document and the
client location, the end user is able to retrieve them more quickly. This reduces the user-perceived latency while accessing the web pages, and reduces the volume of network traffic from the web content provider [2, 9, 13]. As shown in the Figure 1, the backbone network traffic is very heavy without the caching, as indicated by the thickness of the line. In the Figure 2, with caching technology, the backbone network traffic volume is reduced and the response time is improved in the user perspective.

The main purpose of this thesis is focused on the client-side web caching latency and throughput performance evaluation. The followed sections provide more detailed descriptions of the web caching technique and how the HTTP protocol relates to the web.

Figure 1: The network traffic volume (without caching implemented).
Caching:

From the physical form of the caching system implementation, the caching can be distinguished as hard caching and soft caching. Hard caching stores the cached web documents on the local hard disk, while soft caching stays alive for a session on locally allocated memory. If the requested web document is in the hard disk but not in memory, a swap-in request is performed that loads the cached object from the hard disk into memory. In a similar manner, if a web object needs to be swapped out physical memory in order to release the available memory space, and if the web object has been modified, then a swap-out request is performed that stores the in-memory web object to the hard disk.
Caching can be implemented in many different network nodes. It can be installed on a local node that serves multiple end users, or a regional node that serves multiple local server nodes. By placing a caching system on those network nodes, the distance that a cached web content has to travel over the Internet is reduced and end users experience quicker response time and faster performance. This caching system design optimizes both network nodes systems and the network resources; it reduces the amount of network traffic that must flow out of and into a local area networks (LAN’s) and/or wide area networks (WAN’s). Thus, the caching system reduces the amount of network bandwidth required to the Internet that a network node need offer [13, 16, 17]. The caching system on each network node typically allocates between 5-50 GB of hard disk space, and requires 64-512 MB of RAM for soft caching [17].

Today’s commercial web browsers also implement caching as a built-in feature (as called browser cache). The browser cache resides on the local host’s hard disk and is managed by the web browser software on that host. More specifically, the web browser software stores or reads recently accessed web pages to or from the user’s hard disk, instead of accessing the Internet when user requests the cached web page again. Typically, the capacity of a browser cache is small, usually it allocates between 5-50 MB of hard disk space [17]. The user can manually adjust the cache’s capacity on the local hard disk by configuring the appropriate setting(s) within the web browser software.

Caching Policies:

The widely accepted caching performance evaluation metric is the “hit rate” (also called the Document Hit Ratio, or DHR), which is the percentage of the number of accessed caching objects in the hard disk and the total number of requests received by the caching service provider in a given period of time. A higher hit rate means a more desired caching performance tuning, and greatly depends on the caching capacity and the number of users who are accessing the Internet with that client. Benchmarks have shown that cache hit rates are typically limited to the 30 to 40 percent range [13, 16].

The original Least Recently Used (LRU) caching policy, which is quite successful in most environments such as the Operating System memory management and the databases searching engine, is not the optimal solution for the network caching implementation strategy. Decreasing the network bandwidth volume is one of the advantages provided by the caching technique, and most of the network bandwidth saving depends on the cache hit document size. The majority of the cache hit web pages sizes
are smaller than the size of a TCP socket output buffer; thus, a high hit rate of the smaller size web documents does not reduce the total number of the network traffic packets much. A variation of the DHR is the Byte Hit Ratio (BHR), which is the percentage of the total number of bytes of all hit web documents transmitted in the network and the total number of bytes of replies from the same server sending to the network. While the caching of large web documents does not greatly improve the user-perceived latency, it does result in a higher BHR. Consequently, the available network bandwidth is used more efficiently when large documents are cached on the local host [7, 9, 14].

**The Internet Infrastructure:**

The modern network infrastructure is shown on the Figure 2. Typically, a client connects to the Internet through an Internet Service Provider (ISP, or server) that generally contains a caching system implementation. Every time the client requests a web page, the URL request is sent directly to the server through the network link. The server processes the URL request by checking its local caching system before it passes the request to the original web page provider. If the requested web content is not cached, the server sends a regular GET request to the original web content provider to retrieve the entire document from the provider. It then retransmits the response from the content provider to the client, and saves a copy of the web page in its local cache system. If, on the other hand, the requested web page is found in the server’s cache, the server sends an HTTP HEAD request to the web content provider to verify the freshness of the document. If the response of the HTTP HEAD request from the provider indicates that the copy residing on the server’s cache system is still up-to-date, then the server sends the copy it already has to the client without modifying it. Otherwise, the server sends a regular GET request to the web content provider, it retransmits the response from the provider to the client, and it saves a copy of the web content on its local cache.

**HTTP Elements:**

The HTTP plays a main role in the exchange of information on the Internet. An HTTP request consists several elements: a method such as GET, and HEAD; a web page location indicator (as called Uniform Resource Identifiers, URI); and a set of name-value pairs of the Hypertext Request Headers (HTRQ). For example: An HTTP request for the web page /index.html on the host www.csc.calpoly.edu has the following format:
GET /index.html HTTP/1.0

User-Agent: Mozilla/4.04 [en] (X11; U; SunOS 5.5.1 sun4m; Nav)
Host: www.csc.calpoly.edu
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8

The “GET” method retrieves the information indicated in the next token, which is referred to as the “Request-URI.” The Request-URI is a relative path for the particular host, which is identified by the tag Host, to locate the document directory and name. Every document has a unique name inside the specified directory on every host. The next token indicates the version of HTTP accepted by the client system. The remainder of the HTTP request is the set of name-value pair tags. The “User-Agent” tag value identifies the client agent that originated the request. It is used for statistical purposes and to prevent server responses that exceed a particular user agent limitation. The HTTP “Host” tag specifies the Internet host name and the port number of the resource being requested. Next, the “Accept” tag field specifies the type of media response accepted by the client system. The value */* means the client browser will accept all the different kinds of media type responses. The “Accept-Language” HTTP header field specifies the language preferred by the client browser. In here, “en” means the language preferred is English. The last header, “Accept-Charset,” is used to indicate that what response character sets is acceptable by the client browser.

The other method, “HEAD”, is almost identical to the method GET except the HEAD method must not include the requested document content in the response message body. This method usually used to obtain web content information such as testing hypertext links for validity, checking accessibility, and the last modified date. It can also accompany the HTTP header to obtain more specific web content attributes, such as the header If-Modified-Since, If-Unmodified-Since, and Pragma.

The HTTP header “If-Modified-Since” is a conditional request used with the GET and HEAD methods. If the requested web page has been modified since the time stamp specified in the value of the If-Modified-Since tag, the server must include the web content inside the response message body of the GET request. Otherwise the server should send the client the status code “304 Not
Modified”, which means the requested web page has not been modified and that the cached copy on the client system is still valid.

The tag “If-Unmodified-Since” also is a conditional request used with the GET and HEAD method. It has the opposite effect to the If-Modified-Since tag. If the requested web page has not been modified since the time stamp specified in the value field of this tag, then the server should perform the requested operation specified by the method. Otherwise, the server must send the client the status code “412 Precondition Failed”, which means the conditional requested examine is failed.

The “Pragma” directive is used as a cache directive. If the value of “no-cache” is specified in thePragma directive, then the server must send the entire web content to the client – even if the requested content is currently sitting in the client’s cache. In other words, even if the requested content is sitting in the client’s cache and the content is up-to-date, it is ignored; the specified content is always obtained from the remote server.

The response from the server consists of the following components: the status code, a set of name-value pair HTTP headers, and the web page body, if applicable. The status code indicates the status of the response from the server. The most popular status code is the 200 OK, which means the requested operation was successful – i.e., the requested information was returned to the client according to the specific requested operation. The status code 304 Not Modified indicates the requested web content has not been modified since the specific time stamp provided by the HTTP request. If the request method is GET, then no web content body should be transmitted to the client. The other two frequently accessed status codes are 403 Forbidden and 404 Not Found. The status code 403 Forbidden indicates that the server understood the request but has not been honored because the client who requests this web resource does not have the suitable authorization to access the web resource. The status code 404 Not Found indicates that the server was unable to find or locate the requested web document. The user should carefully reexamine the HTTP request to verify the request accuracy.

Outline of the Thesis:

The main purpose of this thesis as mentioned before is focused on the client-side web caching latency performance evaluation. This thesis investigates the performance such as how the caching
implementation saves browser latency from the user’s point of view, how much time it saves, and is there any limitation of the caching performance (i.e. the CPU speed, memory size, disk I/O performance, etc.).

The following chapters describe in more detail this web caching simulation, and illustrate the performance analysis and results. Chapter 2 previews some of the relevant works pertaining to the study of caching systems. Chapter 3 provides the motivation behind this thesis, the simulation test environment, and the simulation software structure. Chapter 4 unveils the latency and throughput performance measurements obtained through the web caching simulation. Chapter 5 provides an analysis of the data collected in Chapter 4, and compares the two test systems’ performance. Chapter 6 suggests some of the possible future works.
CHAPTER 2: Related Works

As the number of the web users will have tripled to more than 175 million from 1997 to 2001, and the number of web pages is growing by more than a million pages per day to 1.5 billion pages by the year 2000. This suggests that network traffic is doubling every 90 days while graphics, audio, video files become more common web objects [13]. The ISDN lines, cable modems, and xDSL connections will increase the bandwidth capacity for the clients, thus those advanced techniques pour more requests to the Internet server, which in turn will consume the server’s capacity more quickly and saturate the network bandwidth. In order to alleviate the demands on the network bandwidth and improve the server’s QoS quality, more and more corporations and organizations are seeking for the enterprise commercial caching solutions. In the past few years, several different approaches to the caching enhancement solution have been proposed. This chapter will explore some of the research approaches.

Cache Manager:

Generally, web pages can be categorized into two types: static pages and dynamic pages. The static pages are the files stored at a server’s data repository. Every time when a client requests a static page, the server directly sends the file to the client without modified it. The dynamic pages are the data constructed by the server’s program at each time a request is made, and the result depends on each of the request. Because the server constructs a dynamic web page each time it receives a request for that page, the response message body is unpredictable and hence slows down the server’s performance. The purpose of the Cache Manager, known as the Dynamic Web cache [4], is to reduce the CPU load on a server as it produces dynamic pages; it is not concerned with reducing the amount of traffic on the network. The Dynamic Web cache has its own API for accessing and reconstructing the contents of the cache. Web applications that generate dynamic web pages can use this API to modify and control the dynamic pages that are stored in the cache. Thus, the web application programs are responsible for ensuring their dynamic web pages are up to date within the cache.

Dynamic Web has been elected as part of IBM’s caching solution and fulfilled at numerous IBM and customer web sites managing the dynamic web pages. In 1996, Dynamic Web has proved its ability of handling a large scale, high-volume Atlanta Olympic Games official web site’s caching system. The performance data collected shows that caching reduced server load significantly, and that the average CPU
time required to fulfill the request from a cache was considerably less than the average CPU time required to fulfill the request of creating a fresh dynamic web page. The average cache hit ratio also achieved around 80%.

From a simulation result on an IBM RS/6000 Model 590 workstation with a 66 MHz POWER2 processor, Dynamic Web achieves the performance of handling 500 requests per second and in near-optimal performance when the server program is executed by a CGI program. Hypothetically, the optimal performance would be achieved when the cache process does not consume any CPU cycle.

**Squid Proxy Performance:**

As the WWW is becoming today’s global information distribution, exchange, and sharing center, the web traffic obtains the exponential growth rate. Caching proxies are the major components to alleviate the web traffic bust, which are deployed between clients and servers connection path and shared by several clients. They store those frequently accessed web documents in a place nearing clients. Thus, while accessing those cached web contents, the clients experience quicker response time and caching proxies reduce the outbound network bandwidth demand at the same time.

Squid is one of the many caching proxies, and uses two-level caching system architecture: one is the physical memory buffer containing the recent frequently requested objects, the other is the local hard disk storing a copy of those cached documents. In the Alex Rousskov and Valery Soloviev study [9], it unveils several interesting result related to the web characteristics and the caching proxy performance. The web traffic pattern is one of the important web characteristics. After analyzing the logs obtained from different levels (root, top-level, and leaf) of caching servers, the traffic patterns were identical to each other and about 99% of transfers are smaller than 64K bytes. This finding convinced that the majority of the popular web documents have smaller file sizes from 100 bytes to 100,000 bytes.

The hit rate is another important performance check mark for the caching proxies. As mentioned in the Chapter 1 Introduction, the hit rate is normally between 30 to 40 percent. For a decent hit rate, caching proxy should contain as many the popular small web documents as possible. For a better Byte Hit Rate, caching proxy should also allow the large file stored on the hard disk. Thus, the total hard disk capacity available for the caching proxy and the upper bound cacheable web documents size limitation are two of the caching hit rate enhancement factors. To find the optimal configuration for these two factors is
an important task. A poor configuration will waste hard disk storage space, decrease the essential hit rate, barely improve the response time from the user’s perspective, and will not significantly reduce the network load.

In the Squid proxy performance analysis, each caching proxy level allocates different hard disk capacity: the root has 26GB hard disk space, the top-level has 6GB hard disk space, and the leaf is assigned 6GB hard disk space. For reducing the network bandwidth, the Squid proxy also allows caching large files (up to 8MB). The result shows that the hit rate on the root is between 30 to 40 percent, on the top-level is between 20 to 30 percent – which is slight less than the root performance – and on the leaf it increases up to 60 percent. The Byte Hit Rate in this analysis is usually about 7 to 9 percent lower than the Document Hit Rate measured on the same level of the caching proxy. The response time enhancement, which depends on the Document Hit Rate, is about 28 percent for the root caching proxy and 60 percent for the leaf level.

Overall, the caching proxies have achieved the significant improvement for reducing the network bandwidth and response time as promised. After deploying a carefully designed caching policy, the hit rate increased while the response time decreased. Allowing caching large file sizes has the effect of improving the Byte Hit Rate and alleviating the network bandwidth usage. Thus the introduction of caching proxies provides a feasible way to solve the web traffic bottleneck by deploying a caching service at that location, thereby increasing the utilization of web resources.

**Delta-Encoding Technique:**

One of the alternative ways to reduce network traffic volume is applying a well-understood compression technique to the cached web objects. This reduces the required transfer size compared to the transfer size before compression applied. Thus, the potential consequence is decreasing the network bandwidth usage. The main compression technique used in the Internet infrastructure is called the Delta-Encoding [17]. The delta-encoding technique has two variants: simple delta and optimistic delta.

Clients and servers that are classified as “delta-encoding enabled” means that a set of special HTTP name-value pair headers are added into the standard HTTP protocol to allow them inform the party that the delta-encoding option is applicable for the following web objects. In a case, if the client and server, which are delta-encoding enabled, store a same version of cached web document on both side, then when the client issues a request of fetching this web document the procedure performed is described as follows.
The client issues a URL request containing the special HTTP header that indicates the delta-encoding option is on, and the version of the requested web object on the client system. The server then sends a URL request to the original web content provider to download the latest version of the required web document. Once the server receives the newest version of the object, it applies the delta-encoding scheme extracting out the difference, the delta, between the copy of the newest version and the old version that client contained. Then the server sends out the delta to the client to reflect the changes between these two versions. Since the client receives the delta, the client starts reassembling the delta with the old copy to obtain the updated version. This scheme described above is called Simple Delta.

The other possibility is that when a client does not have a copy of the desired web pages and the server has a copy, the server first sends the copy it owns to the client. In the meanwhile, the server also issues a URL request to the original web page provider to obtain the newest version of that object. Once the updated version received, the server produces the delta and sends it to the client at the end of the current transmission. Once the client receives both transactions, it reconstructs the required object from them and displays it in the browser. This procedure is called Optimistic Delta.

The experiment in [17] indicates that, considering the simple delta strategy, for the mean value of the original file size 54339.25 bytes, the average delta size generated is about 3392.59 bytes. Thus the average percentage of the delta size generated and the original file size is about 6.24%, which implies that the network bandwidth saving is approximately 93.76%. The average latency for fetching the delta is 7.75 seconds, and fetching the original file is 3.93 seconds. Thus the benchmark for the latency ratio of delta to original file is 1.97, which represents that it takes 1.97 times longer to transfer the delta encoded file than to transfer the original non-delta file. This degradation is the result of the extra computation time required, and might also be due to the experimental implementation that deploys the client and server on the same computer. On the trial of applying optimistic delta-encoding scheme, the average latency for transferring the optimistic-delta is about 8.34 seconds and the mean latency for transferring the original file is around 3.95 seconds. Thus the benchmark for the latency ratio of optimistic-delta to the original non-delta file is about 2.11, which implies that it take 2.11 times longer to transfer the optimistic-delta encoded file than to transfer the original non-delta file.
The original delta-encoding pioneers, Jeffery C. Mogul et al., state that:

“Simple deltas benefit by trading off computation of the deltas for reduction in bandwidth and latency over the slow link when both sides store the same old version of a page. Optimistic deltas trade off an increase in the amount of data transferred, by sending an older version during an idle time of the slow link followed by a delta, for a reduction in end-to-end latency” [18, 19].

This statement perfectly explains the outcome reported in [17].
CHAPTER 3: Motivation and Implementation Scheme

As mentioned in the Chapter 1, the section outline of the thesis, this thesis is interested in the client-side web caching performance. In other words, it focuses on the caching performance of the end user’s system and does not consider the caching performance on the Internet server end. The primary reason is that based on the behavior of the average web clients, the set of regularly accessed web pages will stay the same for a while. Thus the client system is a good candidate of implementing a web caching system. This caching system will honor those frequently accessed web pages retrieve with the faster response time as well as save the network bandwidth because the cached web pages are already resided on the client’s hard disk. Based on this study, the thesis also tries to determine if there is any performance limitation caused by the physical system barrier, and will the physical barrier overwhelm the benefit obtained from the caching performance.

Client Systems:

The thesis selects the Linux kernel 2.2.12-20 as the client operating system because of its Open Source policy. The Open Source policy reveals the kernel source code to the public. Any software developer who is interested in interaction with the operating system can obtain a copy of the source code and modify the coding to fit the special purpose as necessary. Thus, the Open Source policy maximizes the flexibility of the operating system to adapting different environment and optimizes the performance.

For a more reasonable result comparison, the client-side web caching system is implemented in two different client platforms. The first client system platform has the following physical configuration: Intel Celeron 266MHz CPU, 96MB RAM, 3Com 905C-TX-M Etherlink 10/100 Mbps Ethernet card, and Linux OS kernel 2.2.12-20, the second client system platform has the following setting: Intel Pentium III 500MHz CPU, 256MB RAM, 3Com 905C-TX-M Etherlink 10/100 Mbps Ethernet card, and Linux OS kernel 2.2.12-20. These test system configurations are randomly selected.

The user-perceived latency measurement requires a web browser interacted with the client caching system. In this thesis, Netscape Communicator, which comes with the Redhat Linux 6.1 distribution, is the first choice. Recall in the Chapter 1 Introduction, the modern commercial web browser has implemented the feature of browser caching. For more precise client-side web caching system performance data gathering, the browser caching interference with the client-side web caching system must be minimized and
the feature of the browser caching must be disabled. Netscape allows the user manually to specify the browser caching configure, hence it also allows the user disable the browser caching. The detail configuration steps are provided in the Appendix.

**Software Modules:**

In here, this thesis uses the software implementation to simulate the caching performance. According to the different functionality, the software package has been separated into different modules. The Figure 3 shows the communication interaction between the software modules. A software module called “Browser Monitor” (or the “Client Proxy”) was created to detect and record user-perceived latency. The Browser Monitor listens on the local host loop back IP address 127.0.0.1 at port 6080 waiting for Netscape to send it a URL request. After a URL request is received by the Browser Monitor, it starts a browser latency timer and sends the URL request to the Server Proxy through the local host look back IP address 127.0.0.1 at port 6020 to fulfill the request operation.

The second software module is the Server Proxy, which plays the major role in this caching performance simulation. The Server Proxy listens on the local host loop back IP address 127.0.0.1 at port 6020 waiting for a client sending a URL request (in this case, the client means the Browser Monitor, not Netscape Communicator). When a URL request arrives, the Server Proxy calls a `fork()` to create a child process, and then the main thread of the Server Proxy goes back to the listening state. The newly created child process is then responsible for executing the URL request. Before the child process sends a

![Figure 3: The communication between the client-side web caching components.](image-url)
URL request to the Internet server, it performs several tasks such as: looking for the Internet server’s IP address through a Domain Name Server (DNS) service; establishing a TCP/IP socket connection between the Server Proxy and the Internet web content provider; searching its local cache repository (the local hard disk), and so on. If there is a match on the cached object for the requested web document, then the child process constructs a HEAD request,

```
HEAD /document_folder/document_name HTTP/1.x
```

and sends it to the web content provider to validate the “freshness” of the cached copy. Note here, that although the Server Proxy holds a copy of the requested web document, it still needs to send a HEAD request to the original web content provider to ensure coherence with the original copy at the provider. After the Server Proxy receives the response from the web content provider, the child process examines the value of the “Last-Modified” tag in the reply message’s header. The value of the “Last-Modified” tag represents the date and time of the requested file’s “last modified” date. If the Last-Modified time stamp value is older than or has the same value as the cached file last modified time stamp, then the child proxy opens the cached document and starts a socket timer before loading and sending the document content to the Browser Monitor through the local host loop back IP address 127.0.0.1 at port 6020. When the file loading finished, the child process stops the corresponding socket timer and reports the time to a log file.

If the Last-Modified time stamp value is newer than the value of the cached file last modified time stamp, the child process constructs a GET request,

```
GET /document_folder/document.html HTTP/1.x
```

and sends it to the original content provider. It also starts a socket timer and prepares to receive the updated web content. While receiving the updated web content, the child process sends the received data to the Browser Monitor through the local host loop back IP address 127.0.0.1 at port 6020, and saves the updated copy to the local cache system. After finishing the file transmission, the child process stops the corresponding socket timer and reports the time to a log file.

If, at very beginning, the child process examines the local cache repository and does not find a matched file, it then performs the same tasks as the Last-Modified time stamp validation fail described above. The tasks include sending a GET request to the original server, starting a socket timer, receiving the
response from the server, transmitting the response data to the Browser Monitor, saving a copy of that data, then stopping the corresponding socket timer, and reporting the time logs to a file.

Figure 4: The flow chart of the software modules communication time frame.
Since the Browser Monitor sends a URL request to the Server Proxy, it waits until a response is sent back from the Server Proxy. While receiving the response, the Browser Monitor forwards the message to the Netscape Communicator application through the local host loop back IP address 127.0.0.1 at port 6080. This process continues until the Server Proxy disconnects, thereby indicating the fulfillment of the request. At this point, the Browser Monitor stops the corresponding browser timer and records the time to a log file for further performance analysis purposes.

Figure 4 shows the software modules’ communication time frame in detail. The circled numbers indicate different stages in the communications process, which are also illustrated in the Figure 3 based on the infrastructure diagram.

**Caching Policy:**

The metrics for evaluating the caching performance are Document Hit Rate and Byte Hit Rate. A higher Document Hit Rate indicates a better latency enhancement. A higher Byte Hit Rate represents a better bandwidth utilization. As previously mentioned in Chapter 1, about 90% of web content sizes are between 100 bytes to 100,000 bytes. Thus, in an attempt to achieve a significant latency improvement, the caching policy should consider storing small sized files – especially files whose sizes are from 100 bytes to 100,000 bytes. However, the strategy of caching small sized web documents will not help reduce network bandwidth much. To alleviate the network bandwidth saturation problem, a higher Byte Hit Rate is desired, which means the caching strategy should consider storing some larger sized web pages. However, caching large sized documents will not achieve a significant latency improvement rate. Therefore, finding the balance point for both latency performance enhancement and network bandwidth optimization will be a tremendous work, and will depend on several Internet parameters such as the client and server system characteristics, the end user group profile, and the network loading pattern.

To simplify the caching strategy issue, this caching simulation implementation does not consider setting a limitation for the cacheable web document sizes at this stage. The main interest of the implementation is investigating the latency and throughput performance boost obtained by applying the caching installation.
Client-Server Geography:

The thesis is more interested in the real life simulation, the widely Internet connection which is not limited by the small-scaled local area network characteristics. Hence, the original content provider is located at the Michigan State University, East Lansing, Michigan, and the clients run on computers that are located in Engineering East (building 20), room 114, California Polytechnic State University, San Luis Obispo.
CHAPTER 4: Performance Results

This chapter reveals the test data collected from the client-side web caching performance simulation. The destination server of the web content provider is located at the Michigan State University, East Lansing, Michigan. Therefore, these simulated data illustrate the real life web surfing latency. For retrieving the test data file, initiates the URL request on the Netscape URL field as the following

http://www.cse.msu.edu/~smithh/test_data_file_name.txt

Test Data Collection:

From the previous researchers reports (see Chapter 1), the majority of the web documents sizes are within 100 bytes to 100,000 bytes, and the average size of the HTML document is around 10K bytes. In this simulation test, the test data files sizes are selected from 100K bytes to 5M bytes in attempt to reflect the more and more audio and video files are dominated today’s web traffic and try to determine the limitations related to the file’s size. Those test data files are separated into two groups: one is the small set of data files and the other is the large set of data files. The small set of data files contains the file sizes from 100K, 200K, 300K, 400K, 500K, 600K, 700K, 800K, 900K, until 1000K bytes. The large set of data files contains the file sizes from 1000K, 2000K, 3000K, 4000K, until 5000K bytes.

Two tests were performed on all test data files: one is the cache hit latency test and the other is the cache hit missed latency test. All the data were collected within a 5 day period from March 27, 2000 to March 31, 2000. Each day, every test data files are retrieved 10 times separately for the case that cache existing in the local system and the case that cache not existing in the local system.

In order to ignore the unpredictable Internet interference caused by other web users and the server’s unusual heavy loading, the valid test data for the performance analysis will only consider 10 of the median range data. That is, first sort all 50 of the collected data values for each test data file from the 5 days simulation period, and then discard the 1st to the 20th and the 31st to the 50th data entries in the collection. The remaining 10 data entries, from the 21st to the 30th, are considered as the valid and reasonable test data results.

The following performance analyses are based on the 10 valid test data collections, and all the performance values are the mean value of the 10 valid test data entries.
Terminology:

Before diving into the performance analysis, the terminology used in this thesis should be given to avoid any confusion caused by different usage of the same technique terms.

a) w/o cache socket latency (sec): Without cache socket latency, the latency timer is started immediately after a web content request is sent, and it is stopped when the TCP connection is closed by the server. The unit measured is second.

b) w cache socket latency (sec): With cache socket latency, the latency timer starts after the cached document is opened, and before loading the first byte from the local cache system. The timer stops after the end of file (EOF) reached. The unit measured is second.

c) % socket latency time saving: The percentage socket latency time saving is the percent difference between a) and b), as defined by:

\[
\frac{(a - b)}{a} \times 100 \quad (\%) 
\]

d) w/o cache socket latency std (sec): Without cache socket latency standard deviation, is the standard deviation of the a). The unit measured is second.

e) w cache socket latency std (sec): With cache socket latency standard deviation, is the standard deviation of the b). The unit measured is second.

f) w/o cache socket latency std %: Without cache socket latency standard deviation percentage, is the percentage of the d) and the a), as defined by:

\[
\frac{d}{a} \times 100 \quad (\%) 
\]

g) w cache socket latency std %: With cache socket latency standard deviation percentage, is the percentage of the e) and the b), as defined by:

\[
\frac{e}{b} \times 100 \quad (\%) 
\]

h) w/o cache browser latency (sec): Without cache browser latency, the latency timer starts before the Browser Monitor sends a URL request to the Server Proxy, and stops after the TCP connection between the Browser Monitor and the Server Proxy is closed by the Server Proxy. In this case, the Server Proxy experiences a cache hit missed. The unit measured is second.
i) w cache browser latency (sec): With cache browser latency, the latency timer starts before the Browser Monitor sends a URL request to the Server Proxy, and stops after the TCP connection between the Browser Monitor and the Server Proxy is closed by the Server Proxy. In this case, the Server Proxy experiences a cache hit. The unit measured is second.

j) % browser latency time saving: The percentage browser latency time saving, is the percent difference between h) and i), as defined by:

\[
\frac{(h - i)}{h} \times 100 \quad (\%)\]

k) w/o cache browser latency std (sec): Without cache browser latency standard deviation, is the standard deviation of the h). The unit measured is second.

l) w cache browser latency std (sec): With cache browser latency standard deviation, is the standard deviation of the i). The unit measured is second.

m) w/o cache browser latency std (%): Without cache browser latency standard deviation percentage, is the percentage of the k) and the h), as defined by:

\[
\frac{k}{h} \times 100 \quad (\%)\]

n) w cache browser latency std (%): With cache browser latency standard deviation percentage, is the percentage of the l) and the i), as defined by:

\[
\frac{l}{i} \times 100 \quad (\%) \quad // \text{‘l’ is a lower-case L, not 1 (one)}\]

o) w/o cache socket throughput (bits/sec): Without cache socket throughput, is the ratio of the file size transferred and a). The unit measured is bits per second.

\[
\text{file}\_\text{size} / a \quad \text{(bits / sec)}\]

p) w cache socket throughput (bits/sec): With cache socket throughput, is the ratio of the file size transferred and the b). The unit measured is bits per second.

\[
\text{file}\_\text{size} / b \quad \text{(bits /sec)}\]

q) % throughput increasing: The percentage throughput increasing, is the percent difference between o) and p), as defined by:

\[
\frac{(o - p)}{o} \times 100 \quad (\%)\]
r) socket latency diff (sec): Socket latency difference, is the time difference between the a) and the b). The unit measured is second.

\[ a - b \text{ (sec)} \]

s) browser latency diff (sec): Browser latency difference, is the time difference between the h) and the i). The unit measured is second.

\[ h - i \text{ (sec)} \]

**Celeron 266 Performance Analysis:**

Figure 5 shows the latency measurements associated with the “small set” test data files on client system comprised of an Intel Celeron 266 MHz CPU, 96MB RAM, and a 10/100Mbps Ethernet card. The top-most line is the browser latency associated with the cache hit missed. The second top-most line is the

![Latency Measurement - Celeron 250MHz, 96MB RAM (small set data files)](image)

Figure 5: Latency measurement of small set ASCII data files – Celeron 266MHz, 96MB RAM.

![Latency Measurement - Celeron 266MHz, 96MB RAM (large set data files)](image)

Figure 6: Latency measurement of large set ASCII data files – Celeron 266MHz, 96MB RAM.
socket latency associated with the cache hit missed. The third line indicates the browser latency when the up-to-date copy of cache resides on the local hard disk. Finally, the last line indicates the socket latency when the up-to-date copy of cache resides on the local hard disk.

Figure 6 illustrates the latency measurement of the “large set” test data files on the same system. The top-most line is the browser latency associated with the cache hit missed. The second top-most line is the socket latency associated with the cache hit missed. The third line indicates the browser latency when the up-to-date copy of cache found on the local hard disk. The last line indicates the socket latency when the still valid copy of cache found on the local hard disk.

These two figures illustrate that the caching system improves both the socket and browser latency. For example, the mean latency of retrieving a 100K bytes web object, with the cache hit missed, the socket latency is about 1.4 seconds, and the browser latency is about 1.7 seconds. If the cache hit success, the latency for the same sized file is improved. The socket latency and the browser latency are 0.1 second and 0.6 second, respectively. Thus, the web user experiences about 1.1 seconds enhancement of the browser latency. For a large file size, 5M bytes, the socket latency is around 32.4 seconds and the browser latency is around 32.9 seconds, respectively, with no match found inside the local cache system. If a copy of the required document is stored on the local cache, then the socket latency and the browser latency are 24.0 seconds and 24.7 seconds, respectively. The user then experiences the significant time saving, 8.2 seconds.

Figures 5 and 6 also show the inclination that larger size of cached object, then higher browser latency saved. More detailed data are provided in the Table 1 and 2. Table 1 collects the socket latency related test data. The values in the 4th column, the socket latency difference (sec), increase as the file size increases. However, the value of the 5th column, the percentage of socket latency time saving, decreases as the file size increases, and range from 90.24% to 26.14%. This could be caused by the hard disk I/O performance, when the file size larger then the available physical memory capacity, then more time is spent swapping data between the hard disk and memory. The column 6th to column 9th indicate the standard deviation of the collected test data. The smaller standard deviation it has, the more accurately the data will be. In the simulation, all the socket latency standard deviations are between 1.54% to 8.50% and the mean percentage is 5.37%. Thus the standard deviations are indirectly endorsed the data accuracy of the collected socket latency.
Table 1: Socket latency measurement – Celeron 266MHz, 96MB RAM.

<table>
<thead>
<tr>
<th>file size (Kbyte)</th>
<th>w/o cache socket latency (sec)</th>
<th>w cache socket latency (sec)</th>
<th>% socket latency time saving</th>
<th>w/o cache socket latency std (sec)</th>
<th>w cache socket latency std (sec)</th>
<th>% socket latency time saving std</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.422527</td>
<td>0.138786</td>
<td>90.24%</td>
<td>0.062497</td>
<td>0.004630</td>
<td>4.39%</td>
</tr>
<tr>
<td>200</td>
<td>2.148709</td>
<td>0.236652</td>
<td>88.99%</td>
<td>0.143098</td>
<td>0.012917</td>
<td>6.66%</td>
</tr>
<tr>
<td>300</td>
<td>2.803370</td>
<td>0.280182</td>
<td>90.01%</td>
<td>0.121538</td>
<td>0.007936</td>
<td>4.34%</td>
</tr>
<tr>
<td>400</td>
<td>4.014538</td>
<td>0.595601</td>
<td>85.16%</td>
<td>0.357321</td>
<td>0.034492</td>
<td>8.90%</td>
</tr>
<tr>
<td>500</td>
<td>4.277300</td>
<td>1.224846</td>
<td>71.36%</td>
<td>0.254048</td>
<td>0.032006</td>
<td>5.94%</td>
</tr>
<tr>
<td>600</td>
<td>4.959889</td>
<td>1.791109</td>
<td>63.89%</td>
<td>0.421787</td>
<td>0.056239</td>
<td>8.50%</td>
</tr>
<tr>
<td>700</td>
<td>5.843129</td>
<td>2.256739</td>
<td>61.38%</td>
<td>0.232500</td>
<td>0.156359</td>
<td>3.98%</td>
</tr>
<tr>
<td>800</td>
<td>5.947085</td>
<td>2.755707</td>
<td>53.66%</td>
<td>0.212189</td>
<td>0.103978</td>
<td>3.57%</td>
</tr>
<tr>
<td>900</td>
<td>6.957324</td>
<td>3.407952</td>
<td>51.02%</td>
<td>0.371715</td>
<td>0.127194</td>
<td>5.34%</td>
</tr>
<tr>
<td>1000</td>
<td>5.941891</td>
<td>3.503476</td>
<td>41.04%</td>
<td>0.164097</td>
<td>0.065569</td>
<td>2.76%</td>
</tr>
<tr>
<td>2000</td>
<td>13.414742</td>
<td>8.481099</td>
<td>36.78%</td>
<td>0.551098</td>
<td>0.214830</td>
<td>4.11%</td>
</tr>
<tr>
<td>3000</td>
<td>19.431665</td>
<td>13.524052</td>
<td>30.40%</td>
<td>1.153381</td>
<td>0.614135</td>
<td>5.94%</td>
</tr>
<tr>
<td>4000</td>
<td>26.003077</td>
<td>18.693778</td>
<td>28.11%</td>
<td>2.009530</td>
<td>0.288288</td>
<td>7.73%</td>
</tr>
<tr>
<td>5000</td>
<td>32.418524</td>
<td>23.943341</td>
<td>26.14%</td>
<td>2.091528</td>
<td>0.546309</td>
<td>6.45%</td>
</tr>
</tbody>
</table>

Table 2: Browser latency measurement – Celeron 266MHz, 96MB RAM.

<table>
<thead>
<tr>
<th>file size (Kbyte)</th>
<th>w/o cache browser latency (sec)</th>
<th>w cache browser latency (sec)</th>
<th>% browser latency time saving</th>
<th>w/o cache browser latency std (sec)</th>
<th>w cache browser latency std (sec)</th>
<th>% browser latency time saving std</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.681688</td>
<td>0.652221</td>
<td>61.22%</td>
<td>0.100324</td>
<td>0.025320</td>
<td>5.97%</td>
</tr>
<tr>
<td>200</td>
<td>2.352014</td>
<td>0.779561</td>
<td>66.86%</td>
<td>0.130254</td>
<td>0.038077</td>
<td>5.54%</td>
</tr>
<tr>
<td>300</td>
<td>3.042800</td>
<td>1.255788</td>
<td>58.73%</td>
<td>0.151067</td>
<td>0.052335</td>
<td>4.96%</td>
</tr>
<tr>
<td>400</td>
<td>4.818336</td>
<td>1.806136</td>
<td>62.52%</td>
<td>0.343450</td>
<td>0.096787</td>
<td>7.13%</td>
</tr>
<tr>
<td>500</td>
<td>4.489225</td>
<td>2.320351</td>
<td>48.31%</td>
<td>0.253583</td>
<td>0.087463</td>
<td>5.65%</td>
</tr>
<tr>
<td>600</td>
<td>5.750299</td>
<td>3.118649</td>
<td>45.77%</td>
<td>0.349572</td>
<td>0.134882</td>
<td>6.08%</td>
</tr>
<tr>
<td>700</td>
<td>6.345801</td>
<td>3.435312</td>
<td>45.86%</td>
<td>0.279842</td>
<td>0.238214</td>
<td>4.41%</td>
</tr>
<tr>
<td>800</td>
<td>6.616390</td>
<td>3.933240</td>
<td>40.57%</td>
<td>0.393677</td>
<td>0.239655</td>
<td>5.95%</td>
</tr>
<tr>
<td>900</td>
<td>7.185159</td>
<td>4.497624</td>
<td>37.40%</td>
<td>0.359186</td>
<td>0.306707</td>
<td>4.93%</td>
</tr>
<tr>
<td>1000</td>
<td>6.407650</td>
<td>4.460803</td>
<td>30.38%</td>
<td>0.072729</td>
<td>0.049399</td>
<td>1.14%</td>
</tr>
<tr>
<td>2000</td>
<td>13.972820</td>
<td>9.794244</td>
<td>29.91%</td>
<td>0.509497</td>
<td>1.106955</td>
<td>3.65%</td>
</tr>
<tr>
<td>3000</td>
<td>19.781506</td>
<td>15.060417</td>
<td>23.87%</td>
<td>1.331669</td>
<td>1.238645</td>
<td>6.73%</td>
</tr>
<tr>
<td>4000</td>
<td>26.194188</td>
<td>20.441564</td>
<td>21.96%</td>
<td>2.317424</td>
<td>1.204403</td>
<td>8.85%</td>
</tr>
<tr>
<td>5000</td>
<td>32.854951</td>
<td>24.864491</td>
<td>24.32%</td>
<td>2.106696</td>
<td>0.568233</td>
<td>6.41%</td>
</tr>
</tbody>
</table>
Table 2 illustrates the browser latency related test data. It also indicates that the percentage browser latency time saving decreases as the requested file size larger and larger. The percentage browser latency time saving is 61.22% for the file size 100K bytes, and down to 24.32% for the file size 5M bytes. The range of the standard deviations from the column 6th to column 9th are between 1.11% and 11.30%, and the average percentage of the standard deviation is 5.46%.

Figure 7 demonstrates the throughput performance verse the file size. The first line (the top-most line) is the throughput performance of the cache hit success. The second line is the throughput performance of the cache hit missed. From this diagram, the maximum throughput performance occurs at the file size 300K bytes and the throughput is about 8.5M bits/sec for the cache hit success. For file sizes larger than 300K bytes, the throughput degenerates dramatically, as shown in the Figure 7. The throughput drops from
<table>
<thead>
<tr>
<th>file size (Kbyte)</th>
<th>w/o cache socket throughput (bits/sec)</th>
<th>w cache socket throughput (bits/sec)</th>
<th>% throughput increasing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>562379.52</td>
<td>5764270.17</td>
<td>924.98%</td>
</tr>
<tr>
<td>200</td>
<td>744633.24</td>
<td>6760982.37</td>
<td>807.96%</td>
</tr>
<tr>
<td>300</td>
<td>856112.56</td>
<td>8565863.82</td>
<td>900.55%</td>
</tr>
<tr>
<td>400</td>
<td>797103.01</td>
<td>5372726.16</td>
<td>574.03%</td>
</tr>
<tr>
<td>500</td>
<td>935169.36</td>
<td>3265715.60</td>
<td>249.21%</td>
</tr>
<tr>
<td>600</td>
<td>967763.69</td>
<td>2679903.45</td>
<td>176.92%</td>
</tr>
<tr>
<td>700</td>
<td>958390.60</td>
<td>2481456.21</td>
<td>158.92%</td>
</tr>
<tr>
<td>800</td>
<td>1076157.41</td>
<td>2322453.34</td>
<td>115.81%</td>
</tr>
<tr>
<td>900</td>
<td>1034880.70</td>
<td>2112705.81</td>
<td>104.15%</td>
</tr>
<tr>
<td>1000</td>
<td>1346372.66</td>
<td>2283446.76</td>
<td>69.60%</td>
</tr>
<tr>
<td>2000</td>
<td>1192717.68</td>
<td>1886548.07</td>
<td>58.17%</td>
</tr>
<tr>
<td>3000</td>
<td>1235097.46</td>
<td>1774616.07</td>
<td>43.68%</td>
</tr>
<tr>
<td>4000</td>
<td>1230623.59</td>
<td>1711799.51</td>
<td>39.10%</td>
</tr>
<tr>
<td>5000</td>
<td>1233862.47</td>
<td>1670610.63</td>
<td>35.40%</td>
</tr>
</tbody>
</table>

Table 3: Throughput data – Celeron 266MHz, 96MB RAM

8.5M bits/sec of the file size 300K bytes to 3.3 M bits/sec of the file size 500K bytes. For the file size 5M bytes, the throughput only about 1.7M bits/sec, which is slightly higher than the maximum speed of T1 line, 1.544M bits/sec.

The cache hit missed throughput is about 0.6M bits/sec for the file size 100K bytes, and the 1.2M bits/sec for the file size 5M bytes. Although the cache hit missed throughput increases as the file size increased, the maximum throughput is bounded by the T1 line speed, which is the slowest throughput value in the transaction path between the client machine and the server at the Michigan State University.

Table 3 provides all the throughput data for every test file. Surprisingly, for the file size of 100K bytes, the socket throughput for a successful cache hit is 9 times faster than the socket throughput of missed cache hit. However, for the file size 5M bytes, the cached socket throughput only increases about 35.4%.

Pentium III 500 Performance Analysis:

Figure 8 represents the latency measurement of the small set test data files on a client machine comprised of an Intel Pentium III 500MHz CPU, 256MB RAM, and 10/100Mbps Ethernet card. The top-most line indicates the browser latency associated with the cache hit missed. The second top-most line shows the socket latency while the cache hit missed. The third line illustrates the browser latency while the
cache hit succeeded. The last line, the bottom-most line, represents the socket latency associated with the cache hit success.

Figure 9 illustrates the latency measurement of the large set test data files on the Pentium III platform. The first two lines are almost identical, which represent the socket latency and the browser latency of the cache hit missed. The third line shows the browser latency for the cache hit success, and the last line indicates the socket latency for the cache hit success.

From Figure 8, the socket latency for retrieving the document size of 100K bytes while a cache hit missed, is approximately 0.9 seconds, and the browser latency is around 1 second. The socket latency for
retrieving the same file while a cache hit succeeded is about almost nothing, an ε, a small enough amount of time to ignored safely and the browser latency is near 0.6 seconds. The socket and browser latencies for retrieving a 5M bytes file while the cache hit missed and succeeded are about 26.8, 26.9, 3.6, and 4.2 seconds, respectively.

These two diagrams also show an attribute that all the latencies (browser, and socket latency with cache hit succeeded and missed) increase linearly as the file size increased. This can be a suggestion that for a given size of file, it is possible to predict the browser and socket latencies for cache hits and misses simply by interpolating the diagrams of Figure 8, and 9.

Table 4 reports the socket related latency measurement. Column 4 shows the socket latency saved by implementing the caching system is around from 0.9 seconds for the file size 100K bytes to 23.2 seconds for the file size 5M bytes. Column 5 illustrates that the percentages of socket latency time saved are from the almost perfect score 99.61% for the file size of 100K bytes down to 86.52% for the file size of 5M bytes. The next 4 columns provide the standard deviation related information of these collected socket latencies. The calculation shows that the standard deviations are within the range from 9.72% to 0.07% and the mean is 3.26%.

The data in the Table 5 are related to the browser latency varies from the file size 100K bytes to 5M bytes. Interestingly, the column 4 shows that the percentage of the browser latency time saved is increasing from 46.06% for the file size 100K bytes to 84.44% for the file size 5M bytes. The time saved is from 0.48 seconds to 22.74 seconds. Column 6th to Column 9th are the standard deviation data, which are in the range from 8.63% to 0.56% and the average are 3.9%.

Those standard deviation data on the Table 4 and 5 are all in an reasonable range; therefore, both the socket latency and browser latency measured for the Pentium III 500MHz machine are meaningful for the performance analysis. Figure 10 shows the throughput of the Pentium III system verse the test data file size based on the data in the previous two tables. The top line demonstrates the socket throughput with the cache hit succeeded, which is the throughput of loading the cached copy from the local hard disk. The bottom line represents the socket throughput with the cache hit missed, which is the throughput of retrieving the requested document from the test server, the Michigan State University.
<table>
<thead>
<tr>
<th>file size (Kbyte)</th>
<th>w/o cache socket latency (sec)</th>
<th>w cache socket latency (sec)</th>
<th>socket latency diff (sec)</th>
<th>% socket latency time saving</th>
<th>w/o cache socket latency std (sec)</th>
<th>w cache socket latency std (sec)</th>
<th>w/o cache socket latency std (%)</th>
<th>w cache socket latency std (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.934017</td>
<td>0.003687</td>
<td>0.930330</td>
<td>99.61%</td>
<td>0.059452</td>
<td>0.00166</td>
<td>6.37%</td>
<td>4.50%</td>
</tr>
<tr>
<td>200</td>
<td>1.490277</td>
<td>0.053124</td>
<td>1.437153</td>
<td>96.44%</td>
<td>0.089160</td>
<td>0.005164</td>
<td>5.98%</td>
<td>9.72%</td>
</tr>
<tr>
<td>300</td>
<td>1.963255</td>
<td>0.099091</td>
<td>1.864164</td>
<td>94.95%</td>
<td>0.061622</td>
<td>0.005630</td>
<td>3.14%</td>
<td>5.68%</td>
</tr>
<tr>
<td>400</td>
<td>2.635532</td>
<td>0.180166</td>
<td>2.455366</td>
<td>93.16%</td>
<td>0.195270</td>
<td>0.004038</td>
<td>7.41%</td>
<td>2.24%</td>
</tr>
<tr>
<td>500</td>
<td>3.083244</td>
<td>0.263347</td>
<td>2.819897</td>
<td>91.46%</td>
<td>0.081043</td>
<td>0.001418</td>
<td>2.63%</td>
<td>0.54%</td>
</tr>
<tr>
<td>600</td>
<td>3.592227</td>
<td>0.339621</td>
<td>3.252606</td>
<td>90.55%</td>
<td>0.151809</td>
<td>0.006624</td>
<td>4.23%</td>
<td>0.18%</td>
</tr>
<tr>
<td>700</td>
<td>4.183606</td>
<td>0.416986</td>
<td>3.766620</td>
<td>90.03%</td>
<td>0.281284</td>
<td>0.001494</td>
<td>6.72%</td>
<td>0.36%</td>
</tr>
<tr>
<td>800</td>
<td>4.702253</td>
<td>0.489334</td>
<td>4.212919</td>
<td>89.59%</td>
<td>0.282595</td>
<td>0.002238</td>
<td>6.01%</td>
<td>0.46%</td>
</tr>
<tr>
<td>900</td>
<td>5.084652</td>
<td>0.571476</td>
<td>4.513176</td>
<td>88.76%</td>
<td>0.182595</td>
<td>0.002709</td>
<td>3.59%</td>
<td>0.47%</td>
</tr>
<tr>
<td>1000</td>
<td>5.435942</td>
<td>0.711086</td>
<td>4.724857</td>
<td>86.92%</td>
<td>0.055153</td>
<td>0.006170</td>
<td>1.01%</td>
<td>8.69%</td>
</tr>
<tr>
<td>2000</td>
<td>10.940660</td>
<td>1.379723</td>
<td>9.560937</td>
<td>87.39%</td>
<td>0.345764</td>
<td>0.002177</td>
<td>3.16%</td>
<td>0.16%</td>
</tr>
<tr>
<td>3000</td>
<td>16.067074</td>
<td>2.123199</td>
<td>13.943875</td>
<td>86.79%</td>
<td>0.480875</td>
<td>0.001461</td>
<td>2.99%</td>
<td>0.07%</td>
</tr>
<tr>
<td>4000</td>
<td>21.194218</td>
<td>2.865438</td>
<td>18.328780</td>
<td>86.48%</td>
<td>0.316846</td>
<td>0.007473</td>
<td>1.49%</td>
<td>0.26%</td>
</tr>
<tr>
<td>5000</td>
<td>26.805747</td>
<td>3.614374</td>
<td>23.191373</td>
<td>86.52%</td>
<td>0.846443</td>
<td>0.006292</td>
<td>3.16%</td>
<td>0.17%</td>
</tr>
</tbody>
</table>

Table 4: Socket latency measurement – Pentium III 500MHz, 256MB RAM

<table>
<thead>
<tr>
<th>file size (Kbyte)</th>
<th>w/o cache browser latency (sec)</th>
<th>w cache browser latency (sec)</th>
<th>browser latency diff (sec)</th>
<th>% browser latency time saving</th>
<th>w/o cache browser latency std (sec)</th>
<th>w cache browser latency std (sec)</th>
<th>w/o cache browser latency std (%)</th>
<th>w cache browser latency std (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.052551</td>
<td>0.567726</td>
<td>0.484825</td>
<td>46.06%</td>
<td>0.066681</td>
<td>0.041614</td>
<td>6.34%</td>
<td>7.33%</td>
</tr>
<tr>
<td>200</td>
<td>1.658895</td>
<td>0.638123</td>
<td>1.020772</td>
<td>61.53%</td>
<td>0.121841</td>
<td>0.049007</td>
<td>7.34%</td>
<td>7.68%</td>
</tr>
<tr>
<td>300</td>
<td>2.128697</td>
<td>0.712568</td>
<td>1.416129</td>
<td>66.53%</td>
<td>0.135050</td>
<td>0.035644</td>
<td>6.34%</td>
<td>5.00%</td>
</tr>
<tr>
<td>400</td>
<td>2.836725</td>
<td>0.805751</td>
<td>2.030974</td>
<td>71.60%</td>
<td>0.244815</td>
<td>0.040643</td>
<td>8.63%</td>
<td>5.04%</td>
</tr>
<tr>
<td>500</td>
<td>3.232377</td>
<td>0.836391</td>
<td>2.395986</td>
<td>74.12%</td>
<td>0.103649</td>
<td>0.031701</td>
<td>3.21%</td>
<td>3.79%</td>
</tr>
<tr>
<td>600</td>
<td>3.716287</td>
<td>0.917150</td>
<td>2.799137</td>
<td>75.32%</td>
<td>0.160552</td>
<td>0.034982</td>
<td>4.32%</td>
<td>3.81%</td>
</tr>
<tr>
<td>700</td>
<td>4.310969</td>
<td>1.035614</td>
<td>3.275356</td>
<td>75.98%</td>
<td>0.289608</td>
<td>0.029790</td>
<td>6.72%</td>
<td>2.88%</td>
</tr>
<tr>
<td>800</td>
<td>4.832781</td>
<td>1.093873</td>
<td>3.738908</td>
<td>77.37%</td>
<td>0.300331</td>
<td>0.020727</td>
<td>6.21%</td>
<td>1.89%</td>
</tr>
<tr>
<td>900</td>
<td>5.201277</td>
<td>1.161365</td>
<td>4.039913</td>
<td>77.67%</td>
<td>0.181758</td>
<td>0.014988</td>
<td>3.49%</td>
<td>1.29%</td>
</tr>
<tr>
<td>1000</td>
<td>5.622838</td>
<td>1.301769</td>
<td>4.321068</td>
<td>76.85%</td>
<td>0.100097</td>
<td>0.026976</td>
<td>1.78%</td>
<td>2.07%</td>
</tr>
<tr>
<td>2000</td>
<td>11.072112</td>
<td>1.975014</td>
<td>9.097098</td>
<td>82.16%</td>
<td>0.359842</td>
<td>0.017795</td>
<td>3.25%</td>
<td>0.90%</td>
</tr>
<tr>
<td>3000</td>
<td>16.198308</td>
<td>2.701600</td>
<td>13.496708</td>
<td>83.32%</td>
<td>0.493280</td>
<td>0.024864</td>
<td>3.05%</td>
<td>0.92%</td>
</tr>
<tr>
<td>4000</td>
<td>21.314467</td>
<td>3.441623</td>
<td>17.872844</td>
<td>83.85%</td>
<td>0.317459</td>
<td>0.019157</td>
<td>1.49%</td>
<td>0.56%</td>
</tr>
<tr>
<td>5000</td>
<td>26.930994</td>
<td>4.189441</td>
<td>22.741554</td>
<td>84.44%</td>
<td>0.844785</td>
<td>0.024214</td>
<td>3.14%</td>
<td>0.58%</td>
</tr>
</tbody>
</table>

Table 5: Browser latency measurement – Pentium III 500MHz, 256MB RAM
Figure 10: Throughput vs. file size – Pentium III 500MHz, 256MB RAM

Figure 10 unveils the dramatic drop of the socket throughput while the cache hit succeeded between the file size 100K bytes and 200K bytes from 127Mbps to 30.12Mbps. The remainder of the graph continues to decrease at a very slow rate from 30.12Mbps to 1.11Mbps as the file size increases from 200K bytes to 5M bytes. The throughput replying the cache hit missed is increasing as the file size increased. It climbs from 0.86M bits per second to 1.49M bits per second that is about 96% of the T1 speed (1.544M bits per second) utilization.

Table 6 summarizes the throughput information. Column 4, the percentage throughput increasing measurement, introduces an unbeatable throughput bust. These data prove that the enhancement is from 25,234.76% to 641.64%. The smaller file size it is the better throughput improvement achieved.
<table>
<thead>
<tr>
<th>file size (Kbyte)</th>
<th>w/o cache socket throughput (bits/sec)</th>
<th>w cache socket throughput (bits/sec)</th>
<th>% throughput increasing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>856515.92</td>
<td>216996229.69</td>
<td>25234.76%</td>
</tr>
<tr>
<td>200</td>
<td>1073626.13</td>
<td>30118213.99</td>
<td>2705.28%</td>
</tr>
<tr>
<td>300</td>
<td>1222459.83</td>
<td>24220185.71</td>
<td>1881.27%</td>
</tr>
<tr>
<td>400</td>
<td>1214176.02</td>
<td>17761397.82</td>
<td>1362.84%</td>
</tr>
<tr>
<td>500</td>
<td>1297334.88</td>
<td>15189096.66</td>
<td>1070.79%</td>
</tr>
<tr>
<td>600</td>
<td>1336218.45</td>
<td>14133393.32</td>
<td>957.72%</td>
</tr>
<tr>
<td>700</td>
<td>1338558.20</td>
<td>13429707.47</td>
<td>903.30%</td>
</tr>
<tr>
<td>800</td>
<td>1361049.69</td>
<td>13078990.56</td>
<td>860.95%</td>
</tr>
<tr>
<td>900</td>
<td>1416026.14</td>
<td>12598947.67</td>
<td>789.74%</td>
</tr>
<tr>
<td>1000</td>
<td>1471686.05</td>
<td>11250405.19</td>
<td>664.46%</td>
</tr>
<tr>
<td>2000</td>
<td>1462434.62</td>
<td>11596532.29</td>
<td>692.96%</td>
</tr>
<tr>
<td>3000</td>
<td>1493738.06</td>
<td>11303696.27</td>
<td>656.74%</td>
</tr>
<tr>
<td>4000</td>
<td>1509845.75</td>
<td>11167576.78</td>
<td>639.65%</td>
</tr>
<tr>
<td>5000</td>
<td>1492217.32</td>
<td>11066923.96</td>
<td>641.64%</td>
</tr>
</tbody>
</table>

Table 6: Throughput data – Pentium 500MHz, 256MB RAM
CHAPTER 5: Conclusion and Summary

The data collected from the simulation provide evidence that the caching technique did in fact reduce the user-perceived latency, and decreased the network bandwidth usage as well. Recall that, this thesis is also interested in determining the physical characteristics of the system limitation introduced to the caching performance. In the following sections, a comparison between the two test client systems is performed, and a summary is provided.

Celeron vs. Pentium III:

Figure 11: Socket latency comparison of small set ASCII data files – Celeron vs. Pentium III

Figure 12: Socket latency comparison of large set ASCII data files – Celeron vs. Pentium III
Figure 11 compares the socket latency of Celeron system with the Pentium III system for the small set of test data files. The two top-most lines represent the latency associated with a cache hit missed on the Celeron and Pentium III systems, respectively. The last two lines indicate the latency associated with a cache hit succeeded on the Celeron and Pentium III system, respectively. Figure 12 reports the socket latency of the Celeron and Pentium III systems for the large set test data files. Again, the first two lines show the latency of the Celeron and Pentium III system while the cache hit missed, respectively, and the last two lines plot the latency of the Celeron and Pentium III system while the cache hit succeeded, respectively.

![Figure 13: Browser latency comparison of small set ASCII data files – Celeron vs. Pentium III](image)

![Figure 14: Browser latency comparison of large set ASCII data files – Celeron vs. Pentium III](image)
These two figures reveal that for retrieving the same size of test file the Pentium host needs less
time to fetch the file. Specifically, when the local cache has a copy of the requested document, the socket
latency difference between the Pentium III system and the Celeron system increases as the size of the test
file increases. For example, the socket latency needed to fetch the 100K bytes document is 1.28 seconds for
the Celeron system and 0.004 seconds for the Pentium III system; thus the latency difference is about 1.28
seconds. Another example, when fetching the 5M bytes data file, the Celeron system required 23.94
seconds and the Pentium III system only consumed 3.61 seconds; hence the latency difference is about
20.33 seconds. This effort is easily noticed by the end user. However, the socket latency difference between
these two systems, while the cache is not available both of them, is not significant. For instance, the latency
required to fetch a 100K byte file from the web is about 1.4 seconds for the Celeron system, and 0.9
seconds for the Pentium III system. The latency difference between these two is only about 0.5 seconds.
For a 5M bytes document, the time consumed for fetching it is around 32.4 seconds for the Celeron host,
and near 26.8 seconds for the Pentium III. The latency difference is about 3.6 seconds. Since the cache is
not available for either system, the socket latency is dominated by the network latency, which is the time
required for the network to transmit the document from the server to the client. Assume the network
characteristics are stable enough during the caching simulation test; thus, the major part of the latency
difference is contributed by the client systems data processing time and is system dependent.

Figure 13 and 14 focused on the browser latency between the Celeron and Pentium III system for
small set testing data files and large set testing data files, respectively. Both figures show that the first two
top-most lines indicate the browser latency of the caching hit missed for the Celeron and Pentium III
system, respectively. The next two lines show the browser latency with cache hit succeeded of the Celeron
and Pentium III system.

The two diagrams illustrate for fetching the same size of data file the Pentium III system has a
quicker response time in the user-perceived latency, for both the cache hit succeeded and missed. For
example, the response time for the Celeron system, while the cache hit fail, of retrieving a 100K bytes
document is 1.7 seconds and 1.1 seconds for the Pentium system. The browser latency difference is about
0.6 seconds. The response time of fetching a 5M bytes data file is 24.9 seconds for the Celeron machine
with the cache hit succeeded, and 4.2 seconds for the Pentium machine. The latency difference is 20.7
seconds, which is a significant performance enhancement. Thus, in the user perspective, the Pentium system has a better performance.

Figure 15 demonstrates the socket throughput comparison for both systems. Note the unit on the y-coordinate is 100M bits per second. The Pentium system achieves the outstanding throughput measurement for the cache hit succeeded situation. Particularly, for fetching the file size of 100K bytes, the socket throughput for the Celeron machine is 5.76M bits per second, and the Pentium machine is 217M bits per second. Thus, the Pentium throughput performance is about 38 times more efficient than the Celeron system at that point. Comparing loading the 5M bytes data file, while the cache hit succeeded, the throughput of the Celeron system is about 1.67M bits per second, and the Pentium system is around 11.07M bits per second. Thus, the Pentium system is about 7 times more efficient than the Celeron system.

![Figure 15: Socket throughput comparison – Celeron vs. Pentium III](image)
This figure also witnesses that the throughput of both systems associated with a cache hit fail does not differ much. For example, the throughput is about 0.56M bits per second for fetching a 100K bytes data on the Celeron machine, and is 0.86M bits per second on the Pentium machine. Hence, the Pentium machine handles about 1.53 times more data than Celeron can at the same time. Comparing the throughput performance for fetching 5M bytes data file while the cache hit fail, the Celeron reaches about 1.23M bits per second, and the Pentium performs about 1.49M bits per second. Hence, the Pentium system executes about 1.21 times faster than the Celeron machine. Recall that, the throughput performance is limited by the T1 speed – which is the slowest transfer speed between the client and server on the network connection path – while the cache hit missed, and it is system independent. Examining Figure 15, it supports the throughput performance limitation explanation fully.

**Conclusion & Summary:**

Several research papers indicate that the major achievement by caching small size files is the end-to-end latency enhancement, and the majority contribution of the caching large size files is the network bandwidth usage saved [7, 8, 9, 14]. The experimental simulation performed by this thesis supports this statement. The socket latency percentage improvement for the Celeron system is from 90.24% for a small size of file 100K bytes down to 26.14% for a large size of file 5M bytes, and for the Pentium system is from 99.61% for a small size of file 100K bytes down to 86.52% for a large size of file 5M bytes. In other words, the socket throughput percentage increment for the Celeron system is from 924.98% for a small size of file 100K bytes down to only 35.40% for a large size of file 5M bytes, and for the Pentium system is from the outstanding performance 25234.76% for a small size of file 100K bytes down to 641.64% for a large size of file 5M bytes.

Figure 7 and 10 illustrate that for both the Celeron and Pentium systems, when the cached file size is large enough, the throughput verses file size plots become stable horizontal lines. The reason behind this can be the system physical attributes limitation such as the CPU speed, available memory size, disk I/O performance. As expected, the Pentium system produces a better performance than Celeron system, even when the system physical characteristics dominate the performance evaluation. On the other hand, when caching is not available for both systems, the performance should be dominated by the characteristics of the network the client system is connected to. Thus, the throughput measurement shows that when the cache hit
is missed, the ultimate transmission speed is bounded by the slowest network connection speed – i.e., the T1 line speed 1.544M bits per second in this case.

Interestingly, Table 5 shows for the Pentium system the percentage of browser latency time saving is increasing while the cached file size increased. The explanation of this special situation is system dependent. If the lower bound of the physical system throughput (T_M bps) performance (the worst case), is better than the maximum throughput (T_1 bps) of the network link, then in any circumstance the caching system will always conduct a better performance than the non-cache available performance. The equivalent mathematics translation is the following:

- System throughput: \( T_M \) bps
- Network throughput: \( T_N \) bps

Therefore, \( T_M \geq T_M > T_1 \geq T_N \).

Hence, for the Pentium system, the browser latency time saving is increasing while the file size increasing too.
CHAPTER 6: Future Work

In this implementation, some interested findings presented themselves. In Figure 7, the socket throughput measurement of the Celeron system for the cache hit succeeded had an unanticipated spike while loading the 300K bytes cached file. Then the throughput starts decreasing quickly until the size of cached files are large enough. The throughput becomes a steady horizontal line again. This behavior is susceptible to the system physical limitation, especially the disk I/O and the available memory size. One scenario is that while fetching a cached file whose size is larger than the available memory, a series of disk to memory content swapping occurs in order to completely load the full file. The dramatically throughput decreasing zone which the file size is from 300K bytes to 1M bytes is conducted by the content swapping degeneration. The followed steady horizontal zone that the file size starts from 1M bytes reflects the stable content swapping performance.

Figure 10, the throughput measurement of the Pentium system for the cache hit succeeded has the same performance pattern: a dramatic drop zone followed by a stable horizontal zone. However, the corresponding y-coordinate scale and the associated dropping zone starting and ending point differ from the Celeron system observation. This phenomenon strongly suggests that this behavior is system dependent. A more thorough study is recommended, such as finding the relationship between system physical parameters, cached file size, and throughput performance.

Another suggested research dimension would be the interaction between client proxy and server proxy located on different physical hosts. Finding optimal caching policy configurations on the client and server proxy – where the deployed caching strategy may be different – that produces the maximum end-to-end latency reduction as well as achieves the optimal network bandwidth usage utilization.

Continuous research based on this implementation has various extensions. A possible study is one that re-conducts the simulation by setting the test data file size from 100 bytes to 100,000 bytes and/or uses different socket buffer size (in this simulation, the socket buffer size is 1500 bytes) then comparing the evaluation results. Also recall that HTTP/1.1 supports data compression by including the header Content-Encoding into the URL request header field. So another feasible extension is that implements the data compression header functionality into the web caching, and then investigates the latency and throughput benchmarks. A related extension is the consideration of implementing the delta-encoding method. As the
finding in [17], the potential benefit of delta-encoding is reducing the network bandwidth usage. Hence, in the simulation, the main purpose will be investigating the bandwidth usage utilization instead of the latency performance.

As the cost of high-end computer equipment has recently become more affordable, web users use different computer configurations while surfing the Internet. Therefore, a study related to comparing the same computer equipment platform with different CPU speeds and different physical memory sizes will also help generalizing limitation of the caching implementation.

Beside that, there are two possible ways to tune-up the simulation software. One suggestion is that while the Server Proxy issues a URL request to validate the cached file’s freshness, a feature can be added to the Server Proxy that allows it to pre-fetching the cached file to the memory regardless of whether the cached copy is fresh or stale. Thus, the pre-fetching scheme will improve the socket latency as well as the browser latency performance. Another suggestion is that while the Server Proxy is fetching a web page from the Internet, a double buffering technique can be added to the Server Proxy. The double buffering allows the Server Proxy to perform the two tasks separately, forwarding the received data to the Browser Monitor, and while concurrently saving the received data to the hard disk. Thus, the socket latency will also be improved.
Appendix:

Browser Caching Configuration:
For a more precise simulation, we need to minimize the interference from the commercial web browser build-in function, browser caching by disable it.
The following is the configuration steps for the Netscape disable the browser caching configuration:
1. Select “Edit” from the Netscape menu bar.
2. Select “Preference…” menu item.
3. A pop-up window shows up, selects the “Advance” from the left-hand “Category” panel.
4. Select the “Cache” on the left-hand “Category” panel under the “Advance”
5. Set the “Memory Cache” field to 0 Kbytes, and then press the “Clear Memory Cache” push button
6. Set the “Disk Cache” field to 0 Kbytes, then press the “Clear Disk Cache” push button

Another commercial web browser software, Microsoft Internet Explorer, has also provided the manually configuration method. The following is the step by step instruction:
1. Select the “Tools” from the application menu bar
2. Select the “Internet Options…”
3. Click on the “General” tab
4. Click on the push button “Settings…” at the “Temporary Internet files” section
5. Check the “Automatically”
6. Set the “Amount of disk space to use:” to 0 MB

Browser Port configuration:
In the implementation, a commercial web browser, the Netscape, is selected as the URL request initiator. It directly connects to the Browser Monitor through the local host loop back address 127.0.0.1 at port 6080, In order to do so, we need to manually configure the network connection setting.
1. Select “Edit” from the Netscape menu bar.
2. Select “Preference…” menu item.
3. A pop-up window shows up, opens the “Advance” from the left-hand “Category” panel.
4. Select the “Proxies” on the left-hand “Category” panel under the “Advance”
5. Check the “Manual proxy configuration” on the right-hand side panel
6. Click the “View” push button on the right-hand side panel
7. Set the “HTTP proxy” field to this value: “localhost”, and the associated “Port” to the value 6080
8. Click on the “OK” push button to save the configuration
9. Click on the “OK” push button again to close the Preference Dialog

Simulation Software:
This section contains all the simulation programs, the C files, H files, and the UNIX make files.

/* File Name: Readme */
/* Description: This file contains a general description of the */
/* relationship between the simulation software and */
/* source codes. It also provides the simulation */
/* procedure step by step. */

/* collects the browser latency */
BrowserMonitor module: BrowserMonitor.c, BrowserSimulator.c, and
NetProxy.h
Compile command: make -f NetMake BrowserMonitor

/* collects the socket latency and server latency */
ServerProxy module: ServerProxy.c, WebRequest.c, ServerHelperFunc.c,
and NetProxy.h
Compile command: make –f NetMake [ServerProxy]

/* combines the browser latency with the socket and server latency, */
/* and creates summary log. */
DataSummary utility: Summary.c, SummaryUtil.c, and Summary.h
Compile command: make –f SummaryMake

Step by step simulation procedure:
1. Start Netscape.
2. Disable the browser caching (as shown in above, Browser caching configuration).
3. Configure the browser proxy connection (as shown in above, Browser port configuration).
4. Start BrowserMonitor:
   $ BrowserMonitor browser_latency_log
5. Start ServerProxy:
   $ ServerProxy serverproxy_latency_log
6. Issue URL requests from Netscape.
7. Terminate BrowserMonitor by Ctrl-C.
8. Terminate ServerProxy by Ctrl-C.
9. Combine browser latency log with the socket and server latency log:
   $ Summary browser_latency_log serverproxy_latency_log combine_log
10. Create a summary log:
    $ Summary combine_log average_log

# File Name: NetMake
# Description: The UNIX make file compiles and creates the ServerProxy program. The shell command as the following,
# $ make -f NetMake
# For compiling and creating the BrowserMonitor program,
# uses the following shell command format,
# $ make -f NetMake BrowserMonitor
# For clean all the object files, ServerProxy, and BrowserMonitor programs, types the tag "clean" as following,
# $ make -f NetMake clean

ServerProxy: ServerProxy.o WebRequest.o ServerHelperFunc.o
gcc -D_REENTRANT -o ServerProxy ServerProxy.o WebRequest.o

ServerProxy.o: ServerProxy.c NetProxy.h
gcc -c ServerProxy.c

WebRequest.o: WebRequest.c NetProxy.h
gcc -c WebRequest.c

ServerHelperFunc.o: ServerHelperFunc.c NetProxy.h
gcc -c ServerHelperFunc.c

BrowserMonitor: BrowserMonitor.o BrowserSimulator.o
gcc -D_REENTRANT -o BrowserMonitor BrowserMonitor.o

BrowserMonitor.o: BrowserMonitor.c NetProxy.h
gcc -c BrowserMonitor.c
BrowserSimulator.o: BrowserSimulator.c NetProxy.h
   gcc -c BrowserSimulator.c
   clean:
      rm *.o ServerProxy BrowserMonitor

/* File Name: BrowserMonitor.c */
/* Description: The C file, BrowserMonitor.c, listens to the port
  BROWSER_PORT at the address BROWSER_ADDR both task is accepting
  the URL request generated by a web browser. It's main
  task is accepting the URL request generated by a web
  browser then forwarding this request to a process
  defined in BrowserSimulator.c to fulfill the
  request. It takes one pass-in parameter, the log file
  name. The log file records the latency of the web
  browser. */
/* Syntax : Executing this program: */
/* $ BrowserMonitor log_file_name */
extern void browserSimulator(int browser_sockfd, long fd);
#include "NetProxy.h"

int main(int argc, char **argv) {
   int browser_sockfd; /* web browser's socket descriptor */
   int browserMonitor_sockfd; /* browser monitor's socket */
      /* descriptor */
   int browser_len; /* length of the web browser socket */
      /* address structure */
   int browserMonitor_len; /* length of the browser monitor */
      /* socket address structure */
   struct sockaddr_in browser_address; /* web browser socket address*/
      /* structure */
   struct sockaddr_in browserMonitor_address; /* browser monitor */
      /* socket address */
      /* structure */
   FILE *logfd; /* file pointer of the log file */

   /* create a log file to record the browser latency information */
   logfd = fopen(argv[1], "w");

   /* logfd == NULL, log file open error */
   if(logfd == NULL) {
      printf("open the browser log file failed!\n");
      exit(1);
   }

   /* create the browserMonitor socket for the web browser to connect */
   browserMonitor_sockfd = socket(AF_INET, SOCK_STREAM, 0);

   /* set the browserMonitor_address domain, type, and port */
   browserMonitor_address.sin_family = AF_INET;
   browserMonitor_address.sin_addr.s_addr = inet_addr(BROWSER_ADDR);
   browserMonitor_address.sin_port = htons(BROWSER_PORT);
browserMonitor_len = sizeof(browserMonitor_address);

/* bind browserMonitor socket with the correct Internet address */
bind(browserMonitor_sockfd,
    (struct sockaddr *)&browserMonitor_address, browserMonitor_len);

/* create a connection queue */
listen(browserMonitor_sockfd, 100);
signal(SIGCHLD, SIG_IGN);

while(1) {
    /* accept the connection from web browser */
    browser_sockfd = accept(browserMonitor_sockfd,
        (struct sockaddr *)&browser_address, &browser_len);
    if(fork() == 0) { /* child process */
        /* forward the web browser request to the function */
        /* browserSimulator() */
        browserSimulator(browser_sockfd, (long) logfd);
    } else { /* parent process */
        /* close the web browser socket connection */
        close(browser_sockfd);
    }
}

return 0;

/* File Name: BrowserSimulator.c */
/* Function : browserSimulator */
/* in : browser_sockfd, a socket descriptor id, indicated */
/* the id of the web browser socket connection. */
/* fd, the file descriptor id, indicated the browser */
/* log file */
/* out : none */
/* Description: The C file, BrowserSimulator.c measures the time */
/* latency starting from sending the URL request to the */
/* Server Proxy ending with the socket connection */
/* disconnected by the Server Proxy. It also forwards */
/* the URL request which is received from the web */
/* browser to the Server Proxy to fulfill the URL */
/* request. */

#include "NetProxy.h"

void browserSimulator(int browser_sockfd, long fd) {
    char buffer[MAX_BUFFER_SIZE]; /* socket buffer */
    int result;
    int netclient_sockfd; /* Server Proxy socket connection id */
    int netclient_len; /* the length of the Server Proxy socket*/
    struct sockaddr_in netclient_address; /* Server Proxy socket */
    struct timeval tvBrowserStart, tvBrowserEnd;
    struct timezone tzBrowserStart, tzBrowserEnd;

    /* ... */
typedef struct {
    /* the structure holds the browser latency data */
    struct timeval tvBrowserStart;
    struct timeval tvBrowserEnd;
} tlatency;

FILE *logfd; /* the browser latency log file's descriptor */

signal(SIGCHLD, SIG_IGN);

/* create a socket connected to the Server Proxy */
netclient_sockfd = socket(AF_INET, SOCK_STREAM, 0);

/* set the netclient_sockfd domain, type, and port */
netclient_address.sin_family = AF_INET;
netclient_address.sin_addr.s_addr = inet_addr(LOCAL_ADDR);
netclient_address.sin_port = htons(LOCAL_PORT);
netclient_len = sizeof(netclient_address);

/* connect netclient_sockfd to the NetClient */
result = connect(netclient_sockfd,
                 (struct sockaddr *)&netclient_address,
                 netclient_len);

/* check the socket connection with the Server Proxy */
if(result == -1) {
    perror("oops: netclient_sockfd failed!");
    exit(1);
}

/* read the URL request from the web browser */
result = read(browser_sockfd, &buffer, MAX_BUFFER_SIZE-1);

/* start the time latency measurement */
gettimeofday(&tvBrowserStart, &tzBrowserStart);

/* forward the URL request to the Server Proxy */
result = write(netclient_sockfd, &buffer, result);

/* read the response from the Server Proxy */
result = read(netclient_sockfd, &buffer, result);
while(result > 0) {
    write(browser_sockfd, &buffer, result);
    result = read(netclient_sockfd, &buffer, MAX_BUFFER_SIZE - 1);
}

/* end the time latency measurement */
gettimeofday(&tvBrowserEnd, &tzBrowserEnd);

/* calculate the browser latency */
tlatency.browser_sec = tvBrowserEnd.tv_sec - tvBrowserStart.tv_sec;
tlatency.browser_usec = tvBrowserEnd.tv_usec -
                       tvBrowserStart.tv_usec;
if(tlatency.browser_usec < 0) {
    tlatency.browser_sec--;
    tlatency.browser_usec = tlatency.browser_usec + 1000000;
}

/* display the browser latency on the screen */
printf("%ld.%06ld\n", tlatency.browser_sec, tlatency.browser_usec);

/* record the browser latency to the browser log file */
(long) logfd = fd;
fprintf(logfd, "%ld.%06ld\n", tlatency.browser_sec, 
tlatency.browser_usec);
fflush(logfd);

/* close the Server Proxy and web browser socket connections */
close(netclient_sockfd);
close(browser_sockfd);
}

/* File Name: NetProxy.h */
/* Description: The H file, NetProxy.h, collects all the necessary */
/* build-in header files. It also contains the constant */
/* variables definition, and the user-defined structure */
/* type. */

#ifndef NETCPROXY_H
#define NETCPROXY_H
#include <arpa/inet.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#define LOCAL_ADDR "127.0.0.1" /* Server Proxy IP address */
#define LOCAL_PORT 6020 /* Server Proxy port number */
#define BROWSER_ADDR "127.0.0.1" /* BrowserMonitor IP address */
#define BROWSER_PORT 6080 /* BrowserMonitor port number */
#define MAX_BUFFER_SIZE 1500 /* maximum socket buffer size */
#define CACH_TMP "cach_tmp/" /* directory for the caching */
#define MAX_HOST_LEN 256 /* maximum length of the host */
#define MAX_GET_LEN 500 /* maximum length of GET */
typedef struct {
    long browser_sec; /* browser latency: holds the portion */
    long browser_usec; /* browser latency: holds the portion */
    long server_sec; /* server latency: holds the portion */
}
/* of second */
long server_usec; /* server latency : holds the portion */  
/* of microsecond */
long socket_sec; /* socket latency : holds the portion */  
/* of second */
long socket_usec; /* socket latency : holds the portion */  
/* of microsecond */
long size; /* the test data file size */  
int type; /* indicates the caching condition: */  
/* 0 -- cached, 1 -- internet */

} _timestruct;
#endif

/* File Name: ServerProxy.c */
/* Description: The C file, ServerProxy.c, listens to the port */
/* LOCAL_PORT at the address LOCAL_ADDR both defining */
/* in the NetProxy.h header file. The main task is */
/* waiting for a client request a web page, then using */
/* fork() function call to create a child process to */
/* fulfill the URL request. */
#include "NetProxy.h"
extern void web_request(int client_sockfd, struct timeval* tvReqStart, 
                        long fd);

int main(int argc, char **argv) {

  int server_sockfd; /* Server Proxy socket id */
  int client_sockfd; /* client socket id */
  int server_len; /* the length of the Server Proxy socket*/
  /* address */
  int client_len; /* the length of the client socket */
  /* address */
  struct sockaddr_in server_address; /* Server Proxy socket address */
  struct sockaddr_in client_address; /* client socket address */
  FILE *logfd; /* the server latency log file */
  /* descriptor */

  /* open a log file to record the socket latency */
  logfd = fopen(argv[1], "w");

  if(logfd == NULL) {
    printf("open server log file failed!\n");
    exit(1);
  }

  /* create a socket, Server Proxy, for the browser to connect */
  server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* set the server_sockfd domain, type, and port */
  server_address.sin_family = AF_INET;
  server_address.sin_addr.s_addr = inet_addr(LOCAL_ADDR);
  server_address.sin_port = htons(LOCAL_PORT);
  server_len = sizeof(server_address);
  bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

  /* listen for a client connection */
  if(listen(server_sockfd, 10) == -1) { 
    printf("listen() failed!");
    exit(1);
  }

  /* accept a client connection */
  client_sockfd = accept(server_sockfd, (struct sockaddr *)&
                        client_address, &client_len);

  /* send the client a URL request */
  web_request(client_sockfd, tvReqStart, 
               client_len, server_usec, socket_usec, 
               server_sec, socket_sec, size, type);

  /* END of the serverProxy.c */

  printf("ServerProxy.c ended successfully\n");
  close(server_sockfd);
  close(client_sockfd);
  fclose(logfd);
  exit(0);
}
/* create a connection queue */
listen(server_sockfd, 100);
signal(SIGCHLD, SIG_IGN);

while(1) {
    struct timeval tvReqStart;
    struct timezone tzReqStart;

    /* accept the connection from web browser */
    client_sockfd = accept(server_sockfd,
                            (struct sockaddr *)&client_address,
                            &client_len);

    /* start the server time stamp */
    gettimeofday(&tvReqStart, &tzReqStart);

    /* fork process */
    if(fork() == 0) {
        /* forward the URL request to the child process to fulfill */
        /* the request */
        web_request(client_sockfd, &tvReqStart, (long) logfd);
    }
    else {
        /* the parent request (from browser) */
        close(client_sockfd);
    }

    return 0;
}

/* File Name: WebRequest.c */
/* Description: The C file, WebRequest.c, performs the URL request */
/* task. It creates the Internet socket connection, */
/* finds the Host name from the DNS (domain name */
/* service), checks the cached file, compares the last */
/* modified time stamp, and loads the web pages from */
/* either the Internet or cached file to the web */
/* browser. */

#include "NetProxy.h"

extern void parseFileName(char *host, char *getCommand, char *tmpFileName);
extern int appendFile(int fd_out, char web_buffer[MAX_BUFFER_SIZE],
                      int web_result);
extern int loadfile(int client_socket, char *buffer, int result,
                    char *file_out, struct timeval* tvReqStart,
                    _timestruct *tlatencyptr);
extern int isHttpTag(char* num, char* web_buffer);
extern int isModified(char* file_out, char* web_buffer);
extern void printLatency(_timestruct *tlatencyptr, long fd);

/* Function: web_request */
/* in : client_sockfd, an integer, indicates the client socket */
/* id */
void web_request(int client_sockfd, struct timeval* tvReqStart,
    long fd) {
    /* the child process of the Server Proxy */
    char buffer[MAX_BUFFER_SIZE]; /* the web browser request */
    int buffer_result; /* the length of the browser */
    int web_sockfd; /* the Internet socket */
    int web_len; /* the Internet socket addr */
    struct sockaddr_in web_address; /* the Internet socket addr */
    struct hostent *web_hostinfo; /* the Internet host hostenv */
    int web_result; /* the Internet socket */
    char web_buffer[MAX_BUFFER_SIZE]; /* the buffer */
    char web_host[256]; /* the host name string */
    char *web_host_pointer; /* the host name string ptr */
    char *web_http_pointer; /* the http url string ptr */
    size_t web_host_end; /* the length of the host */
    size_t web_get_end; /* the length of the GET */
    char *file_out; /* the Server Proxy log file */
    int fd_out; /* the Server Proxy log file */
    char tmp_file_out[MAX_HOST_LEN + MAX_GET_LEN]; /* cached file */
    int open_test; /* the cached file */
    int tmpfile_exist; /* 0 - the cached file not */
    struct timeval tvServerStart; /* server latency starting */
    struct timeval tvServerEnd; /* server latency ending */
    struct timezone tzServerStart; /* server latency starting */
    struct timezone tzServerEnd; /* server latency ending */
    _timestruct tlatency; /* latency summary struct */
    tmpfile_exist = 0; /* 0 -- file not exist, 1 -- file exist */
    /* read the request information from the browser */
    buffer_result = read(client_sockfd, &buffer, MAX_BUFFER_SIZE-1);
    /* parsing the host information from the browser's buffer */
    web_http_pointer = (char *) strstr(buffer, "http://");
web_host_end = strcspn(web_http_pointer+strlen("http://"), "/ ");
strncpy(web_host, web_http_pointer + 
    strlen("http://"), web_host_end);

web_host[web_host_end] = '\0';

/* construct the GET request */
web_get_end = strcspn(web_http_pointer + 
    strlen("http://")+web_host_end, "\n");

/* create another socket connect to the web page provider */
web_sockfd = socket(AF_INET, SOCK_STREAM, 0);
web_address.sin_family = AF_INET;

web_host_pointer = web_host;
web_hostinfo = (struct hostent *)gethostbyname(web_host_pointer);
if(!web_hostinfo) {
    fprintf(stderr, "no web page provider found!\n");
    exit(1);
}

/* find the internet address */
web_address.sin_addr.s_addr = inet_addr(
    inet_ntoa(*(struct in_addr *)*web_hostinfo->h_addr_list));

web_address.sin_port = htons(80);
web_len = sizeof(web_address);

/* connect the web_sockfd to the internet web page provider */
web_result = connect(web_sockfd, 
    (struct sockaddr *)&web_address, web_len);
if(web_result == -1) {
    perror("oops: web_sockfd failed!");
    exit(1);
}

/* extract the request command */
memset(&web_buffer, 0, MAX_BUFFER_SIZE);
strncat(web_buffer, web_http_pointer+strlen("http://") + 
    web_host_end, web_get_end);
strcat(web_buffer, "\n");

/* get the cached file name */
memset(&tmp_file_out, 0, MAX_HOST_LEN + MAX_GET_LEN);
strcpy(tmp_file_out, CACH_TMP);
parseFileName(web_host_pointer, web_buffer, tmp_file_out);
file_out = tmp_file_out;

/* Save the server response to the cached file */
open_test = open(file_out, O_RDONLY|O_CREAT|O_EXCL, 0600);
if(open_test == -1) {
    /* the requested page already exists in the cach */
tmpfile_exist = 1;
} else {
    /* the requested page not exists in the cach */
tmpfile_exist = 0;
close(open_test);
unlink(file_out);
}

memset(&web_buffer, 0, MAX_BUFFER_SIZE);
if(tmpfile_exist) {
  /* send the HEAD and If-Modified-Since tag */
  struct tm *dc;
  struct stat sbuf;
  char sdate[64];
  char fmt[128];

  strcpy(web_buffer, "HEAD ");
  strncat(web_buffer, web_http_pointer+strlen("http://") + 
         web_host_end, web_get_end);
  strcat(web_buffer, "\n");
  strcat(web_buffer, "\n\n");
  write(web_sockfd, &web_buffer, strlen(web_buffer));
  web_result = read(web_sockfd, &web_buffer, MAX_BUFFER_SIZE);

  if(web_result > 0) {
    if(isHttpTag("304 ", web_buffer) || 
      (isModified(file_out, web_buffer) > 0)) {
      /* load the web page from the cached file */
      loadfile(client_sockfd, web_buffer, web_result, file_out, 
                tvReqStart, &tlatency);
      printLatency(&tlatency, fd);

      close(web_sockfd); /* close the web socket */
      close(client_sockfd); /* close the browser socket */
      exit(0); /* exit */
    }
  }
}

/* no cached file or cached file been modified */
/* send the regular GET request */
if(tmpfile_exist) {
  /* construct the web socket again */
  web_sockfd = socket(AF_INET, SOCK_STREAM, 0);
  web_result = connect(web_sockfd, 
                        (struct sockaddr *)&web_address, web_len);
  if(web_result == -1) {
    perror("oops: web_sockfd failed!");
    exit(1);
  }
}

/* construct the GET method command */
memset(&web_buffer, 0, MAX_BUFFER_SIZE);
strcpy(web_buffer, "GET ");
strncat(web_buffer, web_http_pointer+strlen("http://") + 
        web_host_end, web_get_end);
strcat(web_buffer, "\n\n");

/* use the original request from the browser */
gettimeofday(&tvServerStart, &tzServerStart);
write(web_sockfd, &web_buffer, strlen(web_buffer));
web_result = read(web_sockfd, &web_buffer, MAX_BUFFER_SIZE-1);

if(web_result) {
    char *contentptr;

    /* print the content-length, if any */
    contentptr = (char*) strstr(web_buffer, "Content-Length: ");
    if(contentptr != NULL) {
        int len;
        char clen[10];
        memset(&clen, 0, 10);
        len = strcspn(contentptr + strlen("Content-Length: "), "\t\n ");
        strncpy(clen, contentptr+strlen("Content-Length: "), len);
        tlatency.size = atol(clen);
    } else {
        tlatency.size = -1;
    }

    /* load from socket */
    if(isHttpTag("200 ", web_buffer)) {
        /* if the request response is valid */
        fd_out = open(file_out, O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR);
        tmpfile_exist = 2;
        if(fd_out < 0) {
            printf("cannot open the file %s, fd_out=%d\n", file_out, fd_out);
            abort();
        }
    }

    /* keep reading the web page contents until server of the */
    /* internet web page provider disconnected (close it own */
    /* socket) */
    while(web_result) {
        if(tmpfile_exist == 2) {
            appendFile(fd_out, web_buffer, web_result);
        }
        write(client_sockfd, &web_buffer, web_result);
        web_result = read(web_sockfd, &web_buffer, MAX_BUFFER_SIZE-1);
    }

    /* stop the server time stamp */
    gettimeofday(&tvServerEnd, &tzServerEnd);

    /* calculate the server latency */
    tlatency.socket_sec = tvServerEnd.tv_sec - tvServerStart.tv_sec;
    tlatency.socket_usec=tvServerEnd.tv_usec - tvServerStart.tv_usec;
    if(tlatency.socket_usec < 0) {
        tlatency.socket_usec = tlatency.socket_usec + 1000000;
    }
    tlatency.type = 1;
tlatency.server_sec = tvServerEnd.tv_sec - tvReqStart->tv_sec;
tlatency.server_usec = tvServerEnd.tv_usec - tvReqStart->tv_usec;
if(tlatency.server_usec < 0) {
    tlatency.server_usec = 1000000 + tlatency.server_usec;
tlatency.server_usec--;
}
/* save the latency summary to the server log file */
printLatency(&tlatency, fd);
/* close the cached file */
if(tmpfile_exist == 2) {
    close(fd_out);
}
/* close the Internet and client sockets */
close(web_sockfd);
close(client_sockfd);
exit(0);

#include "NetProxy.h"

/* Function : parseFileName */
/* in : *host, a char pointer points to the Host name string*/
/* getCommand, a char pointer points to the GET method*/
/* string. */
/* *tmpOut, a char pointer points to the parsed file name. */
/* out : none */
/* Description: This function creates the unique caching file name */
/* based on the passed in Host name and the GET method string. For example, if the Host string is the following: "www.csc.calpoly.edu", and the GET method string is "GET /index.html HTTP/1.1", then the generated unique cached file name will be "wwwcsccalpolyeduiindexhtml". The generated unique cached file name will be assigned to the char *tmpOut, and the caller of this function can uses the corresponding parameter obtaining the unique cached file name. */
void parseFileName(char *host, char *getCommand, char *tmpOut) {
    char tmpHost[MAX_HOST_LEN]; /* holds the Host name */
    char tmpGet[MAX_GET_LEN]; /* holds the GET method string */
    char *hostHeadPtr; /* a pointer points to the beginning of the Host name */
    char *getHeadPtr; /* a pointer points to the token next to the GET method */
    size_t tokenLen; /* the length of the token */
    int totalLen; /* the total length of a string */
char subGet[MAX_GET_LEN];  /* a temporary buffer holds the */ /* substring of the GET string */

/* set the pointer to the beginning of the Host and GET strings */
hostHeadPtr = host;
getHeadPtr = getCommand;

/* extract the "GET" from the GET string */
memset(&subGet, 0, MAX_GET_LEN);
tokenLen = strcspn(getHeadPtr, " \
\t"");
strncpy(subGet, getHeadPtr, tokenLen);
getHeadPtr = subGet;

/* clear the temporary buffers */
memset(&tmpHost, 0, MAX_HOST_LEN);
memset(&tmpGet, 0, MAX_GET_LEN);

/* create part of the unique cached file name based on the Host */
/* name */
totalLen = 0;
while(totalLen < strlen(host)) {
    tokenLen = strcspn(hostHeadPtr, ".\n\t\n"");
    strcat(tmpHost, hostHeadPtr, tokenLen);
    hostHeadPtr += tokenLen + 1;
    totalLen += tokenLen + 1;
}

/* create part of the unique cached file name based on the */
/* GET string */
totalLen = 0;
while(totalLen < strlen(subGet)) {
    tokenLen = strcspn(getHeadPtr, ".\n\t\n ");
    strcat(tmpGet, getHeadPtr, tokenLen);
    getHeadPtr += tokenLen + 1;
    totalLen += tokenLen + 1;
}

/* create the unique cached file name */
strcat(tmpOut, tmpHost);
strcat(tmpOut, tmpGet);

/* Function : appendFile */
/* in : fd_out, a file descriptor id, indicates the output */
/* file */
/* web_buffer, a char pointer, holds the data */
/* web_result, a integer, indicates the length of the */
/* data of the web_buffer string */
/* out : a integer, the length of the data record to the */
/* output file */
/* Description: This function appends the data in the web_buffer of */
/* length indicated by the web_result to the cached */
/* file pointed by the fd_out. It returns the length of*/
/* the data succeeded to the caller. */
int appendFile(int fd_out, char web_buffer[MAX_BUFFER_SIZE],

/* */
int web_result) {
    char *endHeaderPtr; /* point to the beginning of the data body*/
    /* that exclude the HTTP header part */
    int headLen; /* the length of the HTTP header part */
    int bodyLength; /* the length of the data body */

    /* extract the HTTP header and calculate the length of the */
    /* message body */
    headLen = headerLength(web_buffer, web_result);
    bodyLength = web_result - headLen;
    endHeaderPtr = web_buffer + headLen;

    /* copy the web_buffer body to the temporary file */
    write(fd_out, endHeaderPtr, bodyLength);

    /* return the length of data copied to the file */
    return web_result;
}

int loadfile(int client_sockfd, char *buffer, int result, char *file_out, struct timeval* tvReqStart, _timestruct *tlatencyptr) {
    struct timeval tvCachStart; /* the loading cached file starting time stamp */
    struct timeval tvCachEnd; /* the loading cached file ending time stamp */
    struct timezone tzCachStart; /* the loading cached file starting time's time zone */
    struct timezone tzCachEnd; /* the loading cached file ending time's time zone */
    struct stat sbuf; /* the cached file attributes struct */
    int open_test; /* the cached file descriptor id */
    int res;

    /* send the web header first, then follows by the raw data */
    len = headerLength(buffer, result);
/* loading the caching file to the browser first */
/* by creating a thread function to do the loading */
res = stat(file_out, &sbuf);
if (res != 0) {
    fprintf(stderr, "%s: stat() 
", strerror(errno));
    return 1;
}

/* open the cached file, and start the socket latency */
open_test = open(file_out, O_RDONLY);
gettimeofday(&tvCachStart, &tzCachStart);

/* send the HTTP header part of the HEAD request to client socket */
write(client_sockfd, buffer, len);

/* load the cached file to the buffer at maximum length */
/* indicate by the constant MAX_BUFFER_SIZE, and send */
/* the data in the buffer to the client socket */
res = read(open_test, buffer, MAX_BUFFER_SIZE);
while (res) {
    write(client_sockfd, buffer, res);
    res = read(open_test, buffer, MAX_BUFFER_SIZE);
}

/* stop the socket time stamp */
gettimeofday(&tvCachEnd, &tzCachEnd);

/* calculate the latency, and save the latency data to the */
/* latencyptr */
tlatencyptr->size = (long)sbuf.st_size;
tlatencyptr->socket_sec = tvCachEnd.tv_sec - tvCachStart.tv_sec;
tlatencyptr->socket_usec = tvCachEnd.tv_usec - tvCachStart.tv_usec;
if (tlatencyptr->socket_usec < 0) {
    tlatencyptr->socket_usec = tlatencyptr->socket_usec + 1000000;
    tlatencyptr->socket_sec--;
}
tlatencyptr->type = 0;
tlatencyptr->server_sec = tvCachEnd.tv_sec - tvReqStart->tv_sec;
tlatencyptr->server_usec = tvCachEnd.tv_usec - tvReqStart->tv_usec;
if (tlatencyptr->server_usec < 0) {
    tlatencyptr->server_usec = tlatencyptr->server_usec + 1000000;
    tlatencyptr->server_sec--;
}

/* close the cached file */
close(open_test);

return res;

/* Function : isHttpTag */
/* in : *num, a char pointer, point to the status code */
/*        number string */
/* *web_buffer, a char pointer, point to the HTTP */
/*        Internet response buffer */
/* out : an integer, 0 - the HTTP response status code is not */
/*        the same code as indicated in *num */
/* 1 - the HTTP response status code is the
/ * Description: This function compares the HTTP response buffer's */
/ * status code with the *num, if they are the same, */
/ * then return 1; otherwise, return 0 */
int isHttpTag(char *num, char* web_buffer) {
    int flag = 0; /* 0 -- not the HTTP *num, 1 -- is the HTTP *num */
    char *headerPtr; /* pointed to the beginning of the status code of*/
    /* the *web_buffer */
    /* point to the status code of the *web_buffer */
    headerPtr = (char*) strstr(web_buffer, "HTTP/");
    /* compare the status code with the *num */
    if(headerPtr != NULL) {
        if(strncmp(headerPtr+strlen("HTTP/1.* "), num, strlen(num)) == 0) {
            flag = 1;
        }
    }
    /* return the result */
    return flag;
}

/* Function : isModified */
/* in : file_out, a char pointer, point to the cached file */
/* *web_buffer, a char pointer, point to the Internet */
/* HEAD response buffer */
/* out : an integer, 0 -- the cached file not modified */
/* 1 -- the cached file been modified */
/* Description: This function compares the last modified time stamp */
/* between the cached file and the file located at the */
/* original web content provider. If the file has been */
/* modified since stored the cached file, then return */
/* 1; otherwise, return 0 */
int isModified(char* file_out, char* web_buffer) {
    const int TOKEN_SIZE = 20; /* the length of the token at the */
    /* last modified string */
    int flag = 0; /* holds the compares result, */
    char *lmptr; /* point to the last modified */
    char token[TOKEN_SIZE]; /* holds the token in the string */
    struct tm dc1; /* cached file timestamp */
    struct tm dc2; /* server responded timestamp */
    struct tm *dc1ptr; /* cached file timestamp struct */
    struct tm *dc2ptr; /* server responded timestamp */
    struct stat sbuf; /* cached file attributed struct */
    /* compare the last modified date of the requested web page with */
    /* the cached file */
    lmptr = (char*) strstr(web_buffer, "Last-Modified: ");
    if(lmptr == NULL) {
        /*...*/
lmptr = (char*) strstr(web_buffer, "Last-modified: ");

if(lmptr != NULL) {
    /* format: "Last-Modified: Tue, 15, Nov 1994 18:45:23 GMT" */
    lmptr += strlen("Last-Modified: Tue, ");

    /* get the day of month */
    memset(&token, 0, TOKEN_SIZE);
    strncpy(token, lmptr, 2);
    dc2.tm_mday = atoi(token);

    /* get the month */
    lmptr += strlen("15 ");
    if(strncmp(lmptr, "Jan", 3) == 0) {
        dc2.tm_mon = 0;
    } else if(strncmp(lmptr, "Feb", 3) == 0) {
        dc2.tm_mon = 1;
    } else if(strncmp(lmptr, "Mar", 3) == 0) {
        dc2.tm_mon = 2;
    } else if(strncmp(lmptr, "Apr", 3) == 0) {
        dc2.tm_mon = 3;
    } else if(strncmp(lmptr, "May", 3) == 0) {
        dc2.tm_mon = 4;
    } else if(strncmp(lmptr, "Jun", 3) == 0) {
        dc2.tm_mon = 5;
    } else if(strncmp(lmptr, "Jul", 3) == 0) {
        dc2.tm_mon = 6;
    } else if(strncmp(lmptr, "Aug", 3) == 0) {
        dc2.tm_mon = 7;
    } else if(strncmp(lmptr, "Sep", 3) == 0) {
        dc2.tm_mon = 8;
    } else if(strncmp(lmptr, "Oct", 3) == 0) {
        dc2.tm_mon = 9;
    } else if(strncmp(lmptr, "Nov", 3) == 0) {
        dc2.tm_mon = 10;
    } else if(strncmp(lmptr, "Dec", 3) == 0) {
        dc2.tm_mon = 11;
    } else {
        printf("impossible month!
");
    }

    /* get the year */
    lmptr += strlen("Nov ");
    strncpy(token, lmptr, 4);
    dc2.tm_year = atoi(token) - 1900;

    /* get hour */
    lmptr += strlen("1994 ");
    memset(&token, 0, TOKEN_SIZE);
    strncpy(token, lmptr, 2);
    strcat(token, ":");
    dc2.tm_hour = atoi(token);

    /* get the minutes */
    lmptr += strlen("12:");
    strncpy(token, lmptr, 2);
}
dc2.tm_min = atoi(token);

/* get the seconds */
lmptr += strlen("45:");
strncpy(token, lmptr, 2);
dc2.tm_sec = atoi(token);

/* get the cached file timestamp */
stat(file_out, &sbuf);
dclptr = localtime(&sbuf.st_mtime);

/* compares the time stamp and get the result */
dc2ptr = &dc2;
flag = tmEquals(dclptr, dc2ptr);
}
/* return the compared result */
return flag;

/* Function : tmEquals */
/* in : *dclptr, a tm struct pointer, points to a time */
/* *struct */
/* *dc2ptr, a tm struct pointer, points to another time*/
/* *struct */
/* out : an integer, 1 - dclptr is newser than dc2ptr */
/* 0 - dclptr is the same as dc2ptr */
/* -1 - dclptr is older than dc2ptr */
/* Description: This function compares two time stamp struct then */
/* return the comparison result. 1 means the dclptr is */
/* newer than the dc2ptr; 0 means the dclptr are the */
/* same as the dc2ptr; -1 means the dclptr are older */
/* than the dc2ptr. */
int tmEquals(struct tm* dclptr, struct tm* dc2ptr) {
    /* compare the year of dclptr with dc2ptr */
    if(dclptr->tm_year > dc2ptr->tm_year) {
        return 1;
    } else if (dclptr->tm_year < dc2ptr->tm_year) {
        return -1;
    }

    /* compare the month of dclptr with dc2ptr */
    if(dclptr->tm_mon > dc2ptr->tm_mon) {
        return 1;
    } else if(dclptr->tm_mon < dc2ptr->tm_mon) {
        return -1;
    }

    /* compare the date of dclptr with dc2ptr */
    if(dclptr->tm_mday > dc2ptr->tm_mday) {
        return 1;
    } else if(dclptr->tm_mday < dc2ptr->tm_mday) {
        return -1;
    }

    /* compare the hour of dclptr with dc2ptr */
    if(dclptr->tm_hour > dc2ptr->tm_hour) {
return 1;
} else if(dc1ptr->tm_hour < dc2ptr->tm_hour) {
    return -1;
}
/* compare the minute of dc1ptr with dc2ptr */
if(dc1ptr->tm_min > dc2ptr->tm_min) {
    return 1;
} else if(dc1ptr->tm_min < dc2ptr->tm_min) {
    return -1;
}
/* compare the second of dc1ptr with dc2ptr */
if(dc1ptr->tm_sec > dc2ptr->tm_sec) {
    return 1;
} else if(dc1ptr->tm_sec < dc2ptr->tm_sec) {
    return -1;
}
return 0;
}

/* Function : headerLength */
/* in : *buffer, a char pointer, point to the data buffer */
/* result, an integer, indicates the length of the */
/* buffer */
/* out : an integer, indicates the length of the HTTP header */
/* Description: This function returns the length of the HTTP header */
/* of the buffer. It returns 0, if the buffer does not */
/* contain the HTTP header part. */
/* Return the length of the socket header */
int headerLength(char* buffer, int result) {
    int length = 0; /* the length of the HTTP header */
    char* endHeaderPtr; /* a pointer point to the beginning of */
    /* the message body -end of HTTP header*/
    int bodyLength; /* the length of the HTTP header */
    int lineLength; /* the length of each HTTP directive line*/

    /* initialize the buffer pointer and total buffer length */
    endHeaderPtr = buffer;
    bodyLength = result;

    /* calculate the HTTP header length */
    if(((endHeaderPtr = (char*) strstr(buffer, "Content-Type: "))
        != NULL) || ((endHeaderPtr = (char*) strstr(buffer,
            "Content-Length: ")) != NULL)) {
        /* the HTTP header part found */
        endHeaderPtr = buffer;
        lineLength = strcspn(endHeaderPtr, "$n");

        /* extract out all the rest HTTP header part */
        while(lineLength > (strspn(endHeaderPtr, "$n") + 1)) {
            endHeaderPtr += lineLength + 1;
            bodyLength = bodyLength - lineLength -1;
            lineLength = strcspn(endHeaderPtr, "$n");
        }
    }
endHeaderPtr += 2;
bodyLength = bodyLength - 2;
} else {
  /* no HTTP header part found */
  endHeaderPtr = buffer;
  bodyLength = result;
}

/* calculate the final HTTP header length and return it */
length = result - bodyLength;
return length;

/* Function : printLatency */
/* in : *tlatencyptr, a _timestruct struct pointer, points */
/* * the latency summary struct */
/* * fd, a long integer, indicates the file descriptor id */
/* * of the log file */
/* * out : none */
/* */
/* Description: This function saves the latency data pointed by the */
/* * *tlatencyptr to the log file which is indicated by */
/* * the file descriptor id, fd. */

void printLatency(_timestruct *tlatencyptr, long fd) {
  FILE *logfd; /* the output log file pointer */
  (long) logfd = fd;

  /* print out the latency summary to the log file */
  fprintf(logfd, "%d \t%ld \t%ld.%06ld \t%ld.%06ld\n",
          tlatencyptr->type, tlatencyptr->size,
          tlatencyptr->socket_sec, tlatencyptr->socket_usec,
          tlatencyptr->server_sec, tlatencyptr->server_usec);
  fflush(logfd);

  /* print out the latency summary to the screen */
  printf("\%d \%ld \%ld.%06ld \%ld.%06ld\n", tlatencyptr->type,
         tlatencyptr->size, tlatencyptr->socket_sec,
         tlatencyptr->socket_usec, tlatencyptr->server_sec,
         tlatencyptr->server_usec);
}

# File Name: SummaryMake
# Description: The UNIX make file compiles and creates the program
# Summary, which generates the full latency summary log
# and the average latency log file
# Syntax : For compiling these files, use the following command,
# $ make -f SummaryMake
# For clean the compiled Summary.* files, use the
# following shell command,
# $ make -f SummaryMake clean

Summary: Summary.o SummaryUtil.o
gcc -o Summary Summary.o SummaryUtil.o

Summary.o: Summary.c Summary.h
gcc -c Summary.c

SummaryUtil.o: SummaryUtil.c Summary.h

gcc -c SummaryUtil.c

clean:
  rm Summary Summary.o

/* File Name: Summary.h */
/* Description: The H file, Summary.h, contains all the necessary */
/* C header files, the constant variables, and the */
/* latency summary structure definition. */

#ifndef _SUMMARY_H
#define _SUMMARY_H

#include <stdio.h>
#include <fcntl.h>

#define SIZE 10 /* set SIZE = 5 for large set of data */
/* set SIZE = 10 for small set of data */
#define COUNT 10 /* number of times run on the same test*/
/* data file of each day */
#define DAYS 5 /* number of days run on the latency */
/* test */

typedef struct {
  int  type;  /* socket connection type: */
  long size; /* size of the fetched data */
  double socket_time; /* socket latency */
  double server_time; /* server processing latency */
  double browser_time; /* browser latency */
}_summarystruct;

#endif

/* File Name: Summary.c */
/* Description: Reads in the latency log files then calculates the */
/* average time latency and output the result to a */
/* file. */
/* Syntax : For combining the server latncy log file and browser*/
/* latency log file into a full report log file uses */
/* the following command, */
/* $ Summary server_log browser_log report_log */
/* For creating the average summary file, uses the */
/* following command, */
/* $ Summary report_log average_log */

#include "Summary.h"

extern void initSummary(_summarystruct *, int size);
extern void scanSummary(long i, long j, _summarystruct *st);
extern void scanAverage(long i, _summarystruct *st);
extern void average(_summarystruct *st, int size, int count);
extern void outputSummary(long l, _summarystruct *st, int size);

int main(int argc, char **argv) {
FILE *fd1, *fd2, *fd3;
_summarystruct summarytime[SIZE], tmp;
int i, j;
int i1;
long l2;
float f3, f4, f5;
int count; /* number of testing times */
int flag; /* 0 - for creating summary output */
    /* 1 -- for creating average output */

if(argc == 4) {
    /* create the full summary output */
    fd1 = fopen(argv[1], "r");
    fd2 = fopen(argv[2], "r");
    fd3 = fopen(argv[3], "w");
    count = COUNT;
    flag = 0;
} else if(argc == 3) {
    /* create the average output */
    fd1 = fopen(argv[1], "r");
    fd3 = fopen(argv[2], "w");
    count = DAYS;
    flag = 1;
}

/* initialize the latency summary struct */
initSummary(summarytime, SIZE);

/* save the latency summary to the report log file */
if(flag == 0) {
    for(i = 0; i < COUNT; i++) {
        for(j = 0; j < SIZE; j++) {
            scanSummary((long)fd1, (long)fd2, &tmp);
            summarytime[j].type += tmp.type;
            summarytime[j].size += tmp.size;
            summarytime[j].socket_time += tmp.socket_time;
            summarytime[j].server_time += tmp.server_time;
            summarytime[j].browser_time += tmp.browser_time;
        }
    }
} else if(flag == 1) {
    for(i = 0; i < DAYS; i++) {
        for(j = 0; j < SIZE; j++) {
            scanAverage((long)fd1, &tmp);
            summarytime[j].type += tmp.type;
            summarytime[j].size += tmp.size;
            summarytime[j].socket_time += tmp.socket_time;
            summarytime[j].server_time += tmp.server_time;
            summarytime[j].browser_time += tmp.browser_time;
        }
    }
}
/* calculate the average latency */
average(summarytime, SIZE, i);

/* save the average latency to the log file */
outputSummary((long) fd3, summarytime, SIZE);

/* close all the opened log files */
if(flag == 0) {
    close(fd1);
    close(fd2);
    close(fd3);
} else if(flag == 1) {
    close(fd1);
    close(fd3);
}
return 0;

#include "Summary.h"

/* Function : initSummary */
/* in : *summarytime, a pointer of the _summarystruct structure */
/* size, the size of the summary latency structure */
/* out : none */
/* Description: This function initializes the summary latency structure to the default value. */
void initSummary(_summarystruct *summarytime, int size) {
    int i;
    for(i = 0; i < size; i++) {
        summarytime[i].type = 0;
        summarytime[i].size = 0;
        summarytime[i].socket_time = 0.0;
        summarytime[i].server_time = 0.0;
        summarytime[i].browser_time = 0.0;
    }
}

/* Function : scanSummary */
/* in : (long) i, a file descriptor id, indicates the file of server and socket latency. */
/* (long) j, a file descriptor id, indicates the file of browser latency. */
/* *st, a pointer of the _summarystruct struct */
/* out : none */
/* Description: This function reads in the latency from two files, */
/* server latency file and browser latency file, and */
/* combines them and save them into the summary latency*/
/* structure. */
void scanSummary(long i, long j, _summarystruct *st) {
    int i1;
    long l2;
    float f3, f4, f5;
    FILE *fd1, *fd2;

    (long) fd1 = i;  /* the server latency file */
    (long) fd2 = j;  /* the browser latency file */

    /* reads in the internet connection type, file size and latency */
    /* data */
    fscanf(fd1, "%d %ld %f %f", &i1, &l2, &f3, &f4);
    fscanf(fd2, "%f", &f5);

    /* display the internet connection type, file size and latency */
    /* data on screen */
    printf("%d 	%ld 	%10f 	%10f 	%10f
", i1, l2, f3, f4, f5);

    st->type = i1; /* internet connection type, */
    /* 0-cached, 1-internet */
    st->size = l2; /* file size */
    st->socket_time = f3; /* socket latency */
    st->server_time = f4; /* server latency processing latency and*/
    /* socket latency */
    st->browser_time = f5; /* browser latency */
}

/* Function : scanAverage */
/* in : (long) i, a file descriptor id, indicated the report*/
/* log file */
/* *st, a pointer of the _summarystruct struct */
/* out : none */
/* Description: This function reads in the latency data from the */
/* report log file to the _summarystruct strucue. */
void scanAverage(long i, _summarystruct *st) {
    int i1;
    long l2;
    float f3, f4, f5;
    FILE *fd1;

    /* the summary log file */
    (long) fd1 = i;

    /* reads the latency data from the report log file */
    fscanf(fd1, "%d %ld %f %f %f", &i1, &l2, &f3, &f4, &f5);

    /* display the read-in latency of the report log file on screen */
    printf("%d 	%ld 	%10f 	%10f 	%10f
", i1, l2, f3, f4, f5);

    /* save the read-in data into the _summarystruct structure */
    st->type = i1;
    st->size = l2;
    st->socket_time = f3;
    st->server_time = f4;
average(_summarystruct *st, int size, int count) {
    int i;

    for(i = 0; i < size; i++) {
        st[i].type /= count;
        st[i].size /= count;
        st[i].socket_time /= count;
        st[i].server_time /= count;
        st[i].browser_time /= count;
    }
}

void outputSummary(long l, _summarystruct *st, int size) {
    FILE *fd;
    int i;

    (long) fd = l;

    for(i = 0; i < size; i++) {
        /* save the average summary latency of each testing data file */
        /* to the report log file, pointer by the file descriptor, fd */
        fprintf(fd, "%d \t%d \t%10f \t%10f \t%10f\n", st[i].type,
                st[i].size, st[i].socket_time, st[i].server_time,
                st[i].browser_time);
        fflush(fd);
    }
}
**Bibliography:**


